First HK/R Mini-Conference on Advanced Object-Oriented Concepts

Sponsor: KIPLING Information Technology AB

Papers

Session 1: Frameworks

Object-Oriented Framworks Definitions by PerOlof Bengtsson.

Extending the power of Object-Orientation by Drazen Milicic.

Encapsulation of a frameworks internal interface in C++ by Charlie Svahnberg.

Session 2: Reuse

Does C++ support reuse? by Ulf Kristiansson.

Well Documented Classes by Riquard van der Vliet.

Session 3: Distribution

Decoupled Message Handling and Broadcasting Messages by Thomas Lidström.

Identification in distributed systems by Magnus Östvall.

BLoB by Andy Eskilsson.

Language Support for Small Applications to become a Big Cooperating System by Magnus Broberg.

Session 4: Design

Developing with Supervisor Objects by Martin Walfisz.

Designing reusable software for home system networks by Mats Bäckström.

Class Hierarchy Changes and Versioning by Erik van Campen.

Session 5: Dynamic Behaviour

Implementing dynamic behaviour in object-oriented languages by Björn Nilsson.

A description of a Dynamic and Interactive Language by Magnus Egnerfors.

CHIT CR by Magnus C. Ohlsson.

Session 6: Foundations

Pursuing the Minimal Type System by Henrik Hermansson.

Garbage collection in C++ using smart-pointers by Jonas Matton.

Definition Modelling by Magnus Robertsson.

Object Oriented Frameworks Definitions[1]

PO Bengtsson
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: PO.Bengtsson@ryepk.ericsson.se
www: http://www.pt.hk-r.se/~mpt95pob

Abstract
The work done concerning object oriented frameworks is in its beginning and most of it tend to concentrate on object oriented frameworks that has been built and how these were built and documented. But there is one question that remains unsatisfactorily answered, i.e. what is a object oriented framework? This is still one of the most common questions and there still exists no answer that is generally agreed on. In this paper some important characteristics of object oriented frameworks are presented, existing definitions discussed and an improved definition is suggested.

1 Introduction

As of today no general definition of object oriented frameworks[2] exists, and it is causing problems for those who wants to develop their own framework and have a share of the benefits associated with it. When starting off, the development teams first of all has to find out what they are going to build and then how. There is still very little people in the industry that really understands the concepts of object-orientation and even less people has any knowledge in object-oriented frameworks. This demands an extraordinary effort when trying to develop a framework for the first time, even if it is in a well known domain for the developers. The existing material provides very little support the first times of framework development. A reason for that might be that they are written by people that are very experienced object oriented designers and programmers, who easily understand the concepts and design issues provided in framework development. There exists no good starting literature and no good explanation of what an object-oriented framework is and what it is not.

Without having a proper knowledge of what an object-oriented framework really is, and the concepts and characteristics which are special to frameworks, there can be no good textbooks or efficient design methods for frameworks. In best cases heuristic approaches can be found.

Many papers and books present their own definitions of a framework or provides one which is incorporated and modified without critically viewing what the new modification does to the definition, i.e. the characteristics are given little thought. Some documentation of framework development has been published, explaining how some frameworks was designed and how they work. But the definitions and explanations provided in these publications are not very critically examined and does only serve the most simple purpose as a definition for the current paper or book in which it is written.

---

1. This paper is part of the proceedings of the first Advanced Object Oriented Concepts (AOOC) mini conference at the University of Karlskrona/Ronneby, January 1996.
2. **framework**, “a basic structure which supports and gives shape, or a broad outline plan etc. thought of as having a similar function”, New Webster Dictionary and Thesaurus. In this paper it is meant software frameworks in particular.

These problems can only be solved when the OO community have agreed on a definition of frameworks that are easily understood and captures the characteristics and core concepts of frameworks.

By studying frameworks to find the characteristics of frameworks, a greater understanding of the problems with frameworks could be developed. The result of such a study would be a a definition that has been carefully founded and evaluated against the existing definitions.

In this paper three important framework characteristics are presented, discussed from a real world perspective and an analysis and design perspective. Existing frameworks definitions are compared and evaluated against the proposed characteristics. Finally an improved definition of object-oriented frameworks is presented.

2 Framework characteristics

There is three major characteristics of object oriented frameworks, they are reusable, domain specific and provides specifications of the dynamic behaviour and structure.

2.1 Reuse

The reuse power of frameworks lies in the combination of having a perfect mixture of abstraction and concreteness. A framework provides an abstract design of a solution and an implementation of the general parts of the solution. Take for example the simple design in figure 1 on page 2. The class `File`, implements a method `Copy` which reads from a source file and writes to a destination file. When reusing this simple design, the implementation of the `Copy` method can be used, and the specific implementations for `Read` and `Write` is supplied in the subclass, e.g `DOSFile` or `UnixFile`. This gives the user of a framework the possibility to reuse the general part and specify the parts that are specific to the current problem. The closer to the general solution of the problem an application is, the more of the implementation can be reused directly from the framework.

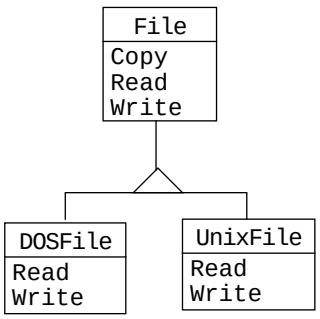


Figure 1. The mix of design and implementation reuse.

2.2 Domain

A domain is a collection of problems and/or solutions that have something in common, e.g. sailing is the domain where different kinds of sailing boats provides a common but slightly varied solution to the problems of water surface transportation. In frameworks the domain concept is important since it provides the knowledge of what is a general solution or problem and what is not. Without it the important mixture of abstract design and generic implementation cannot be achieved. The domain defines the scope of the framework and provides the designers and the users of the framework with important information. The designer must find and decide what is generic and what must be specified for each case. The development

of a framework is very dependent of the domain knowledge. If the domain is new itself and has no general model or set of rules, it is much harder to find generic processes and solutions. But if it is a well defined domain, with stable rules and processes, it is much easier to find the generics parts.

2.3 Dynamic behaviour and structure

Reusable and domain specific only, does not cover the complete concept of a framework. Even standard libraries can achieve those two alone. But those together with the specification of processes, reusable domain specific processes, make up a framework. Frameworks allows for generics processes to be implemented and when later reused, having parts of the process specified to carry out the case specific task. On an object oriented level it is the possibility to specify the behaviour between objects, implementing the generic parts and leaving the case specific implementation to the framework user. This is the main different s between standard component libraries and frameworks, the processes or inter object dynamic behaviour.

The structure is specified when defining the relations of certain classes and objects. The structure is more than data structures like sets and lists, in frameworks the processes in the framework provides structures, which is not found in standard component libraries.

3 Frameworks in the real world

One of the main goals with the object-oriented paradigm is to provide support for easy mapping of real world entities into the implementation. It therefore seems natural to take a look around and see if there exists anything with similar characteristics to object oriented frameworks.

For example building industry. Building can be very modularised. Office buildings and industry buildings are put together from premanufactured standardised modules. Walls, roofs, rooms, doors, etc. conforms to a standard and fits together, thus making an infinite number of combinations possible. The specialisations made are limited to the granularity of the modules. If there is no wall module suited, the traditional way of building must be used for the affected parts.

The reuse of both design and implementation is pretty straight forward, since both the architect and the carpenter has great reuse possibilities. The building frameworks are also domain specific, e.g. industry buildings form a separate domain from office complex frameworks or warehouse frameworks. As for the structure, of course structure is provided by the framework. The structure of a building is to most people very obvious and therefore it is given little thought, but it is there in the shape of room placement etc.

Now, what about the dynamic behaviour of a building? A building does not move or execute any other tasks. True, not directly that is. The building frameworks are constructed to provide support for different domain dependent processes, e.g. a villa framework influences the everyday processes going on in the home, such as dining, laundry etc. A better way of expressing this would be to say that the building framework controls the possible processes and the dynamic behaviour in the building.

There is, however, a big different between building engineering and software engineering, the domain knowledge. Building engineering is a discipline that has thousands of years of experience, and is highly evolved and spread. Almost every person on this planet knows how a house or a building usually look like. The result is that many of the hard earned experiences are now taken for granted. To really learn from other disciplines software engineers must look more carefully, and really examine the things taken for granted. After all, that is what we are aiming for, to reach a state where todays problems are solutions that no one needs to think very much about.

4 Analysis and Design Problems

4.1 Abstraction level

When designing a framework is important to make it general enough to be useful in more then one application and still not make it to abstract forcing the user of the framework to implement a lot classes that were not needed anyway. This offers contradictions in the design work and is possibly one of the hardest thing about object-oriented frameworks. It is very easy, in C++ for example, to end up with a design that is a class called object from which all objects inherit from and this is mainly due to the difficulties of finding a suitable abstraction level. Or put in another way, the problems with finding the balance between generic implementation and abstract design.

4.2 Behaviour specification

When designing frameworks some way of specifying the inter object behaviour is needed. Since more then one type of object is involved the behaviour specification becomes scattered as small pieces of code or design specification in the class definitions, thus making it harder to both understand and reuse. This can put the mixture of generic implementation and design reuse in danger, since it is hard to know how and where the behaviour is defined and were extensions and specialisations are to be implemented. In a normal application development this is not as big issue since no reuse requirement exists.

4.3 Language support

Why is object-oriented frameworks possible at all? What technical facilities are needed to build object-oriented frameworks. The combination of abstract process descriptions and concrete implementation of code, put some demands on the language used. The constructions used in todays object oriented languages offers some support for the implementation of inter object dynamic behaviour. The designers has dynamic binding, polymorphism, inheritance, multiple inheritance, abstract classes, etc. as tools. It is by using this constructions the dynamic behaviour must be constructed. The best case would be if some language could provide possibilities to specify inter object behaviour in a single centralised specification. Then it would be more easy to find, maintain and reuse the dynamic behaviour of systems in general.

4.4 Black box and white box frameworks

This classification of frameworks [Johnson et. al. 88, 91] that are getting widely spread and deals with the reuse mixture of abstract design and generic implementation. A black box framework that has the ultimate level of generic implementation and abstract design would be a framework that only has to be customised by composing classes. A framework consisting only of abstract design, and very little generic implementation would be a white box framework.

5 Existing Definitions

5.1 Birrer and Eggenschwiler

"A framework implements a generic part and defines the places where extensions have to be made in order to support specific solutions" [Birrer]

This definition catches the mixture of design and implementation, i.e. design and implementation of the generic part and design of the solution specific parts. This definition does not demand a framework to be reusable. Since this is one of the important characteristics people expect from a framework, it should be demanded be the definition. Also, it is not always extensions only for an specific solution but also adaptions and customisations that will be made. This is not part of the definition.

The domain part is left out of this definition. This is another of the characteristics and although it might seem obvious for some, it should be emphasized.

Finally, no mention at all about the dynamic behaviour and the structure required in a framework. For all that this definition counts, a simple type parameterised class template, e.g. template in C++, could be a framework, letting the template definition being the generic implementation and the type parameter being the defined place for the specific solution. Many would agree that a C++ template is not a framework, and the framework definition should help determine that.

5.2 Johnson and Foote

"A framework is a reusable design of a program or a part of a program expressed as a set of classes." [Johnson et. al. 88, 91], [Johnson 92]

The reuse concept is included in this definition but only as a requirement that the design framework must be reusable and then it must be expressed as a set of classes. By this definition, the framework does not have to provide any ways for solutions specialisations. The mix of generic implementation reuse and the more abstract design reuse is left unmentioned.

Neither does this definition emphasize that a framework does rely heavily on the domain and are domain specific. One could argue that since it is a program or a part of a program the framework will be part of the same domain as the program. However this is not true, and not the interesting aspect. First, a program can for example be in a administration domain and the framework used is a database framework. It is then a part of that program but cannot be considered to be in the same domain. Second, the domain connection is interesting as characteristic since it is the base for the framework design.

The last part of the definition states how frameworks are documented, as a set of classes. This could be a way of defining the object orientation, but I think that there is more to the specification of a framework then the classes. Therefore, it is not relevant in the definition by its own. Nothing is said about the dynamic behaviour of the framework, nor about the abstract processes that are controlled by the framework. This is also a drawback with this definition.

There is one thing that is good about the definition, it is short.

5.3 Lajoie

"An object-oriented framework, also known as class framework or simply framework, is a set of classes which provide a foundation for solutions to a set of problems. Frameworks are typically designed by experts in a particular domain and then used by non-experts. A framework allows for reusing the abstract design of an entire application, modelling each major component with an abstract class" [Lajoie et al. 94]

The definition acknowledges that it is a solution to a set of problems, not necessarily related at all. The domain emphasis come in the second sentence, but says only that the designers of frameworks are domain experts. The framework does not, according this definition, have to be domain specific.

The reuse aspect is handled, stating that frameworks allows for reuse of the abstract design of an entire application. The word application, brings unclarity to the definition, since it can be interpreted in at least two contradicting ways. First, an application could be a computer program, e.g. a word processor. In that case the definition shuts some frameworks out, since they do not allow for reuse of a complete computer program. Secondly, it can be interpreted as reuse is only allowed when the framework is applicable or relevant, e.g. a database management framework does not in itself provide an abstract design for a complete program but it may very well be completely relevant when used in a program to solve the problem of persistence. Now, that interpretation complies with the characteristics of a framework much more. However, the word application is ambiguous, and many people think of it as the first alternative, a computer program.

The last part of the definition seems to be a throw-in, not given much thought. I consider it as a design guideline for frameworks, but not as a relevant part of the definition. Perhaps it is a way of defining the object orientation, but it misses much of that purpose. If it is a good design guideline, or not, is left for some other discussion.

Again we have a definition which does not incorporate the dynamic behaviour and structure characteristic.

5.4 Gibbs, Nierstrasz and Tsichritzis

"libraries of standard components and program templates that are specific to an application domain and that can be reused to construct a number of similar systems" [Gibbs et. al. 92]

The reuse characteristics of a framework is, in this definition, when it is possible to construct a number of similar systems using the framework. This is not wrong, of course, but it is merely made possible by the mixture of generic implementation reuse and the abstract design reuse. These two are replaced by this weaker statement. Further a framework is said to be made up by libraries of standard components and program templates. I think this is an approach to actually express that it does not have to be specified in a set of classes, as seen in other definitions. Unfortunately, program templates is not really what is about either, rather templates specifying the dynamic behaviour and the structure of the framework solution.

The domain characteristics could be traced to the part, "a number of similar systems', if agreeing that this makes up a domain. Although, it could be explicitly written that it is belonging to a certain domain.

5.5 Keefer

"A framework is an abstract design that describes the interaction between interacting objects in a particular application domain." [Keefer 94]

Here we find a definition that has put emphasis on the object interaction, which is good. It deals with the dynamic behaviour but not the structure. It also has caught the domain connection, in a satisfying way. Unfortunately, the reuse concept is left out. It is perhaps, the abstract design part that is to define the reusability, but the characteristics of mixture, is not covered by that. This definition also benefits from its size.

6 Framework definition

From the characteristics and the above discussed definitions the following definition is synthesized:

An framework is a reusable design and reusable generic implementation of the dynamic behaviour and object structures of a general solution to one or more problems in a certain domain.

This definition fits in the reuse characteristics, stating “reusable design and generic implementation”. The domain connection and emphasis is defined by “to one or more problems in a *certain* domain.”. As for the dynamic behaviour and structure, it is handled in the middle part, “the dynamic behaviour and object structure of a general solution”.

7 Conclusion

Some important concepts has been proposed and discussed. These characteristics are the major characteristics of frameworks and should be handled in a framework definition. The existing definitions of frameworks presented in this paper, all deal with some of these characteristics. Not the same ones, not always directly but the notion of these characteristics can be found. From these experiences, a new definition, is synthesized. Incorporating these major characteristics, in a concise way.

8 Future work

Three major characteristics is certainly not all that exists for frameworks. There must exist more, not as obvious and major, but still important when it comes to getting a complete understanding of what a framework is, how to build them, how to document them, how to use them etc. From this perspective the future work must be to dig even deeper into frameworks.

Acknowledgement

I would like to thank Jan Bosch who provided me with the original idea for this paper.

References

[Birrer]. Andreas Birrer and Thomas Eggenschwiler, ‘Leveraging Corporate Software Development., UBILAB, Union Bank of Switzerland

[Lajoie et al. 94]. Richard Lajoie, Rudolf K. Keller, ‘Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert’, CRIM, May 94

[Huni et. al. 95]. Hermann Huni, Ralph Johnson, Robert Engel, ‘A Framework for Network Protocol Software’, Association for Computing Machinery, Inc., 1995.

[Johnson et. al. 88, 91]. Ralph E. Johnson, Brian Foote, ‘Designing Reusable Classes’, *first published in Journal of Object-Oriented Programming*, July 1988, August 1991

[Johnson 92]. Ralph E. Johnson, ‘Documenting Frameworks using Patterns’, *OOPSLA’92*

[Gibbs et. al. 92]. Oscar Nierstrasz, Simon Gibbs, Dennins Tsichritzis, ‘Component-Oriented Software Development’, *Communications of the ACM*, vol. 35, no. 9, September 1992, pp. 160-165.

[Keefer 94]. Paul Dustin Keefer, ‘An Object Oriented Framework For Accounting Systems’, *Masters Thesis*, University of Illinois at Urbana-Champaign, 1994.

Extending the power of Object-Orientation[1]

Drazen Milicic
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: mpt95dmi@pt.hk-r.se
www: http://www.pt.hk-r.se/~mpt95dmi

Abstract
This paper intends to highlight and emphasise the, often on hand, techniques that can and should be used to create well designed components and systems, reusable components and maintainable systems of high quality. It identifies five larger problems in object-oriented construction and proposes how these can be solved. The main issue it to use the at hand techniques, but in new situation, using other approaches. It is about bringing traditional object-oriented development together with framework thinking.

1 Introduction

Many research papers describe bold new ideas that help to solve different solutions, existing or non-existing. However, most people involved in software engineering can not adopt to those new ides because they are so far fetched from the reality of today. The majority of software engineers need slow transitions that can be incorporated into (not overthrow) there current models, processes and technologies.

The direction of this paper is such that the contents could be incorporated into an organization today. The contents is aimed to be of use by those how feel that they want to go one step further in there object-oriented development techniques. The focus is on small changes that almost any organisation can adopt to in there strive for improvement.

> *If wee can learn to take those small steps that are within reach, we will eventually come up to those huge leaps everybody is talking about.*

Most techniques/technologies that are mentioned herein are closely related to the once already used in the object-oriented world today;

- they are not anything you have to await for twenty years, to successfully try out;
- the problems and scenarios tackled spring from languages (C++, Objective C, Smalltalk, etc.) that exist and are used in the industry.

Many problems within software engineering are not related to the development languages, techniques or technologies, however, there exists much possibilities in improving also these parts to. The paper will discuss available, and some new(?), improvements that could be incorporated into the work to improve the software design. As the title suggests, the problem approach is that it is not enough that a technique exists, but it must also be used and understood to be of any benefit. The paper aims to show why and how techniques can be used, not only in the scope they are presented at research level, but also in the daily work of a software engineer. There are a lot of good ideas floating around in the object-oriented word and all we have to do is to capture them and start using them.

---

1. This paper is part of the proceedings of the first Advanced Object-Oriented Concepts (AOOC) mini conference at the University of Karlskrona/Ronneby, January 18.

For example, the ideas that the frameworks bring to us should not exclusively be used when building frameworks, they could also be incorporated into “normal”[1] object-oriented system development

> *It is about using the obvious, the on hand techniques, without considering them as unreachable and extraordinary occurrences.*

2 Design of today

This section tries to shed some light over the difficulties that exist when developing systems today. Most problems are characteristic for larger systems and it is in such systems they best show there damaging effect. Still, often enough the problems are there also in smaller system, but just not as visible for the viewers. The different sections show what problems that can occur during the design phase of software development. The problems are divided in the larger and more problematic parts that occur during the system design lifestyle.

> *The problem is that much of what is claimed to be object-oriented development actually is not, it is only development in an object-oriented language.*

What ever the cause of the problems are they are still there, requiring attention and new approaches to reach some solutions.

2.1 Finding objects that correctly reflect reality.

One of the first parts that a software engineer will encounter is the need to translate the requirements into some kind of object model, constructing a first “correct”[2] view of the reality. This is one of the more critical actions performed in the software lifestyle. Even if it usually is not considered so, this phase outlines how the rest of the system is divided and organized. If an fault is implanted at this stage, it will only get bigger with time. The idea of how to split up a system in subsystems often is generated at this point. This stage is the foundation for much of the coming work. Now, a counter argument could be that object-oriented development is iterative/incremental so this problem is not so big. Nonetheless, there still is a first time and it is easier to do it right the first time, before everybody has started to work on there separate parts. If a system solution is tried and a full scale development starts of, then it is rather hard to discard the work done because an iteration is needed. However the cards might fall, the real problems do not occur until the software engineers start working on the different parts of the system.

If the work in this phase (i.e identification of potential objects) goes wrong the whole foundation could become distorted. For example, the system could be divided in incorrect parts which could lead to a system that is hard to overview, maintain and understand. It is important to get a correct structure of the system so it reflects the system that actually is intended to build. A system can quickly become a very bad system, because it was build on the perceptions grounded in an early phase of the project. Once the full-scale construction has started, the point of no return might have been passed a long time ago.

2.2 Subsystems

A closely related problem is the problem of dividing the system into subsystems. To divide a system into optimal subsystems (if there are any) can be a hard task. The problem that occurs is of the same nature as the one with the hen and the egg. To get the best possible subsystems you need to have much of informa-

1. Normal indicates the average level of object-orientation that is usually found in the schools and in the industry among those software engineers that use object-oriented techniques.
2. As correct as possible, looked at form the subjective point of view of the software engineer.

tion about details of the system and subsystems, but the main idea with dividing into subsystems is to get better understanding of the system. This can become a issue that causes a lot of different problems.

- If the system is peculiar divided it might become an impossible task to assemble the parts into one whole system again.
- The subsystems could be divided in such way that the same code exists at several places. Data could be spread out on several location within the program. At the worst case these to can exist at the same time yielding that space, speed and functionally gets to suffer.
- The whole system becomes a mess because nobody understands the whole picture. If it is not “spaghetti code” there surely is a risk for “ravioli code”.
- The core functionality of the system gets spread out through out the whole code.

2.3 Object-Orientation does not come with the language!

Object-orientation, and the languages that claims to be object-oriented, often gets blamed when something goes wrong due to the features that object-orientation should help dealing with. There are cases where organisations change development languages from e.g a functional language to an object-oriented language in hope to develop object-oriented systems that are easier to develop, maintain etc. After a while it stands clear that the only thing that happened was that the syntaxes were exchanged, the language was different but noting was new.

The core here is that it takes more than an object-oriented language to write an object-oriented system. Object-orientation is not as trivial as it sometime seems, it is not just an addition to the language - it is a new way of thinking. Further, the new way of thinking is not just about creating and finding the correct objects, but about the whole structure of the system. Finding objects only takes the system design half-way, the rest of the way is making the objects work together and this is where the picture often gets ugly.

The characteristics of an object-oriented language, that separates it from other types of languages, is the notion of polymorphism and inheritance. If these two are not used in combination, the so called object-oriented program could just as well have been developed in another, non-object-oriented, development language. Object-orientation does not put a lot of nice features into the code, it only provides the means to do so, you still have to put in every feature you want your self (neither this is as trivial as might seem?!).

2.4 New design

How many times is a new design directly understandable due to the familiarity of the constructions used? The answer of that question for most software engineers would probably be: not many times, if any!

To better illustrate the question an example, a comparison, between a software system and an car engine is made.

- When you are looking at a new software system, the design documents[1] are examined to get the description of how the system is built up. Mostly the design does not tell what is actually wanted, so looking into the code becomes the only solution.
- A new engine arrives for a check-up (this engine is totally new and no recognizable parts exists in it). Together with the engine a number of books describing maintenance and the parts of the engine are delivered. The books describe nothing of interest, so instead the engine is disassembled in order to understand how it works![2]

1. The design document contains everything above pure code level.
2. Not even a car engineer would understand a engine who’s components where totally unknown down to the smallest parts.

A new design often contains new parts throughout the whole description (perhaps the most fundamental building blocks are familiar). This means that every time a new design is read everything has to be learned all over again from scratch, no knowledge will be for free, everything must be read and understood. This scenario leads to difficulties of reusing components (hardly and rarely knowing what the components really are does not inspire reuse). Even if a design was extremely well-documented it is doubtful (maybe impossible) that everything from all aspects and angles was described. Say that, for reasons sake, such a design existed it still would not be that good because then the material would probably be so extensive that it would be impossible to find or understand everything due to the large amount of information included in the design specification.

2.5 Do we think of tomorrow

Often when designers solve problems they only solve the problems that exist, so that the software will be working at the deadline. There seldom exists any thought of how the system should be expanded, changed or maintained in the future. In those rare cases there exist such thoughts and ideas they are neither documented nor properly expressed in the software system. When the time comes to, in some way, maintain the system, all the knowledge has ben forgotten or the people that had the knowledge have moved on to other projects.

It is hard to se the robustness of a design, the differences between the core functionality and the situation specific and exchangeable parts. Few design documents show how the structure was prepared to be altered, to be maintained. This situation often leads to problematic situations as soon as a system is targeted for maintenance. The moment a module or a system is being changed it rapidly becomes a bad patchwork of changes that fulfils its functions more and more poorly the more it is patched.

If there are any thoughts about maintainability and reusability within the boundaries of a system they must be explicitly expressed and the construction of the software should be constructed in such manner that it becomes obvious where to put in the effort when changing the system.

3 Towards better design

In Section 2 on page 2 a number of problems connected with object-oriented programming are demonstrated. To great extent these problems can be can be, if not removed, helped by using object-oriented techniques that have been in existence for a while in the world of object-orientation. However, many of the techniques are associated with the advanced side of object-oriented development. Such techniques are often thought of when something special and extraordinary should be developed, and not with the daily object-oriented system development. What this section stresses is that techniques may be used in other scopes, such as normal development, with great benefit. In other words, using ideas from abstract design (design by **what** and not **how**), polymorphism, inheritance and programming towards interfaces (not object/classes) will enable the design to get more robust, flexible, understandable and maintainable.

Much of the properties described are collected and captured by the system architecture promoted by framework thinking. Those concepts are too good to use only when developing framework products, when they just as well could be applied when developing all kind of “normal” (non framework[1]) system. The following sections shall show, by thinking in new ways about the same old things, how problems (described in Section 2.1 on page 2 to Section 2.5 on page 4) can be dealt with in more pleasant ways.

1. Non framework products means products who’s intention is not to construct a framework and sell this. A system can be a normal application who’s system architecture is based on concepts normally promoted by frameworks.

3.1 Abstract foundation

One of the first problems previously stated was the difficulty of finding an object model corresponding to the reality, and further the closely related problem of dividing the system into subsystems. As then argued, it is important to divide the system in the "correct" subsystems, as a incorrect start only can lead further from the "reality"/"truth".

This section will show how abstract thinking can be used to divide the system in manageable and more correct parts. To get an object-oriented twist on this section one could say that it is about encapsulation and interfaces of software units (i.e clusters of classes). Though it is important to encapsulate, not only objects, but also larger parts that in some way represent real world functionality (in contrary to the detailed functionality within objects that solve very small parts of the big picture).

One of the problems previously raised about subsystem decomposition was the one that corresponded to the hen and the egg scenario. A partial solution to the problem is one of the most fundamental properties presented by the frameworks. By experimenting with the abstract structure of the system (concentrating on **what** it should do and not **how**) a few designers could stabilize the overall structure of the system. The point is that the modelling should be done with as manageable parts as possible. As can be seen in framework solutions the complete program structure is based on the communication among abstract[1] classes. Instead of having to think of what parts are needed to solve the problems, the effort is spent on specifying an abstract solution to the problem. The solution approach is that the software engineer defines a set of abstract classes with interfaces that support all the actions needed to solve the problem. Hence, no effort is spent on specifying how the interface shall solve the detailed work needed. This setup provides the software engineer with, easy to work with, components that are no trouble changing and/or exchanging. Further, this arrangement allows a few persons to focus on the complete abstraction of the problem and will by that yield a better composition of subsystems. Only when the "over all system constructors" are done with there work will the rest of the project be allowed to participate in the construction work (each person will then be responsible for a clearly defined subsystem). Now, when the full-scale development starts, the whole system (in abstract) will have been iterated and refined a number of times in pursuit of the optimal model.

Usually when a system design is viewed, the thing that is presented is a number of objects that work together according to relationships specified (see Figure 1 on page 5). Usually there has been a number of

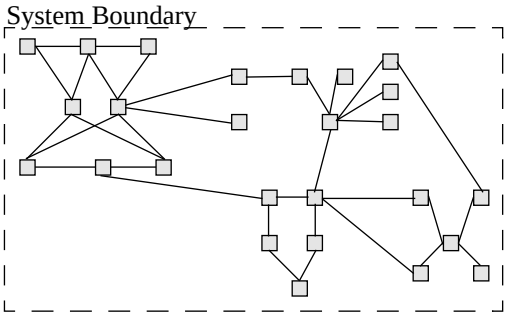


Figure 1. Traditional system construction.

people involved in the models construction work. The problem with such approaches is that a badly structured system will actual not be discovered until the parts are assembled into one system. By then it can be considerably late to start over and begin refining the model.

1. An abstract class can be a class containing code that uses (pure) virtual methods (C++) who's intentions are to be overloaded to get the correct functionality. The classes will still be free from detailed knowledge of how different technical solutions should be constructed.

What frameworks do better, and what could be incorporated into a usual system application, is encapsulating the abstract functionality of the system. Instead of focusing on the whole system at the same time, only the **what**-part is tackled. Due to that the focus can be put on the abstract functionality, the designers are steered into working with the overall parts and how these will interact with each other. The engineers will construct the complete system with abstract classes, only specify interfaces of the functionality provided.

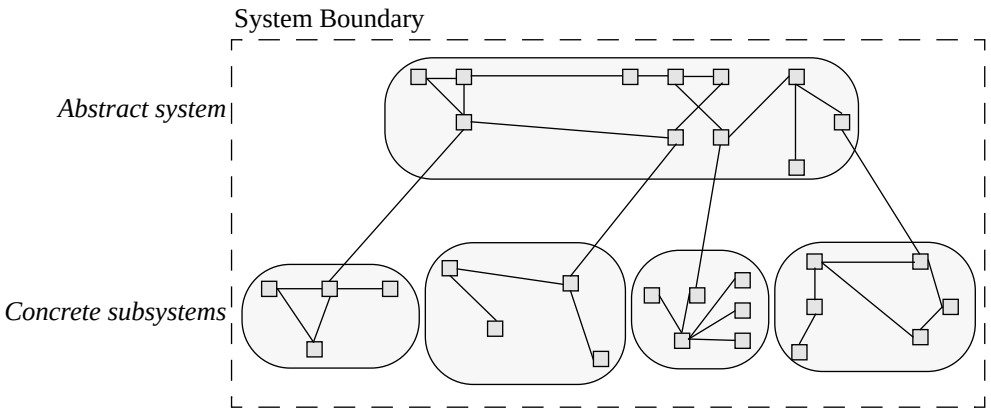


Figure 2. System constructed with framework concepts.

This in turn will spring a good base for system decomposition with a natural separation between abstract and concrete functionality (see Figure 2 on page 6). Further, the system would get an refined subsystem design due to the initial but abstract design efforts.

The benefits with a successful decomposed system are many. Below a few important once are listed.

- The subsystems are (ex)changeable and there by easy to maintain.
- Correctly divided systems, with loos coupling between subsystems, enables the designers to construct the optimal subsystems without any dependencies to worry about.
- As with objects, the subsystems are easier to understand if they can be viewed one at the time without having the detailed knowledge about every other subsystem.

A drawback (or lack of today's development environments) with the proposed ideas in this section is that there is no real language support for encapsulation of such large elements, especially if you want to group a number of classes into one module and give the module a single interface. The nearest possibility is to encapsulate all objects in one large interface object, but this is not really what is needed.

3.2 Object-Oriented construction

Object-oriented programming is very popular today. If object-orientation was as well spread and well used as it was popular then this paper would probably be superfluous. The problem addressed in this chapter is the one that earlier was presented as: object-oriented development is not the same as development in object-oriented languages/environments.

The contribution that development according to framework principles can give is a correct view of object-oriented development, i.e development with abstract interfaces, polymorphism and dynamic binding (C++). Frameworks show how to squeeze the most out of object-orientation. If more systems where constructed according to the principles advocated by frameworks (not only framework products but all kinds of systems), we would have more benefit of the power that is provided by object-oriented concepts. The software systems would be easier to maintain, extend, change and the design would be more stable, easier to divide into subsystems and dynamic in its behaviour, and all this to the prise of learning object-orientation

as it was originally intended to be used. The figures below show a comparison between a simple system build in an traditional way and a framework that solves the same problem, in an much more elegant way.

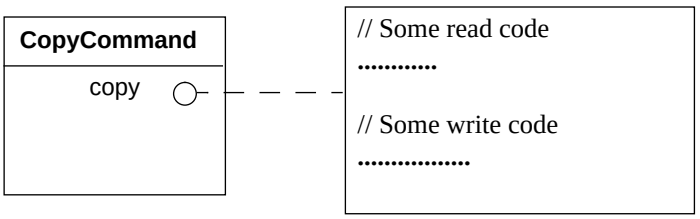


Figure 3. The worst traditional solution to a simple copy command. There is nothing object-oriented with this solution more than that the code is put in an object.

The examples are maybe a bit exaggerated but the important thing is the message that the figures are sending, there is a object-oriented way of doing things and there is a non object-oriented way of doing things, both in an object-oriented language. Some might say that Figure 4 on page 7 takes object-orientation one

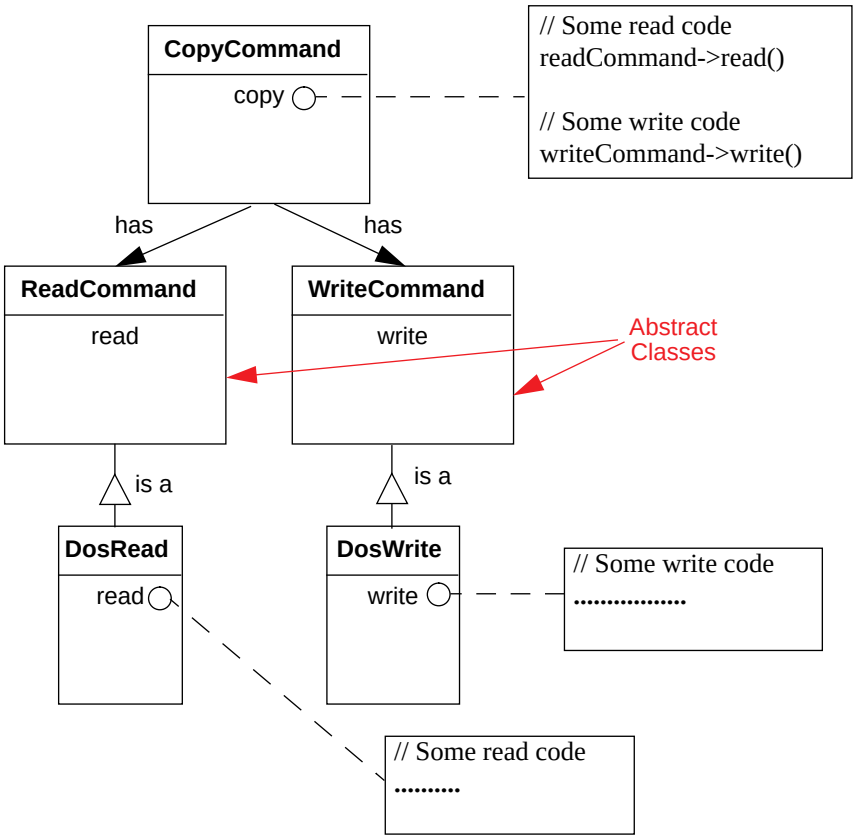


Figure 4. A more object-oriented approach to the copy command.

step to far. However, this is not the point here, that is another discussion.

One of the many benefits with frameworks is that it allows/forces the designer to focus on the important parts. If forces the constructors to reason in more abstract terms that are not so detailed but rather in terms that are, if not the same, very similar as those that where stated by e.g a customer. Because of this, the developers are given the complete picture (of the system) at an earlier stage, which is good for the overall development of software systems. Another thing that a properly constructed framework can do for its users is to help them understand **what** is happening and not **how** the different details are working. It provides the

viewer with necessary abstract information and hides the detailed irrelevant information. One could say that the framework captures the abstract functionality of a program.

One of the stated problems with frameworks is that it is so much harder to construct. To such a statement there actually is two responses, both a true-statement and a false-statement.

- True, because choosing to construct the software according to framework principles will make the development work harder.
- False, because it is not the framework in it self that makes the software much harder to construct.

The true-statement is well established and needs no further discussion. The false-statement however would need some further explanation.

Frameworks forces the constructors to think one more time. Frameworks promote correct use of the object-oriented concepts in languages that support object-oriented concepts. This is why frameworks are considered so hard to construct. They force the software engineers to take full use of the object-oriented facilities within the language and this, perhaps, is not so easy as it seams (there is always better ways, than one currently can see, to use object-oriented techniques).

> *It is not the frameworks concepts in them selves that are so hard. It is rather the more advanced form of object-orientation that they represent that is considered so hard to fully grasp.*

Yet another technique that yet has not been mentioned is the patterns. Patterns could be seen as tiny frameworks, where a good design solution for a problem is captured. Patterns are often design constructions that have evolved out of requirements like flexibility, stability, reuse and extendibility. In pure technical terms one could say that patterns show how object-oriented constellations could be used optimally for a specific problem.

3.3 Robust composition

The problems now remaining to be dealt with are the once regarding recognisability of design and how to make the design stable for tomorrow, i.e a robust design. The issue of how important it was with recognisable design was previous raised. The pro argument for such abilities was that it helped to gain speed and quality in development work. Now, consider the ultimate form of reuse: reuse of requirements. Suppose a requirement is given and recognized. Suppose further that there is a design pattern that solves this particular problem (including implementation). The situation would then be that the complete chain of the software engineering life cycle could be reused, everything throughout the complete chain of development.

The scenario described above is an example where a pattern could have enormous pay back. The scenario might seem far fetched but, it is not an impossible scenario if there has been development of a number of systems in the same domain for a longer period of time.

One of the greatest benefits that a (design) pattern has, is that it is well known. The power that comes with a well-known pattern is that developers recognize it when they read designs. This in turn will increase the understanding of the designs by far. If design specifications are explicitly documented to stress the existence of patterns, designs could become those powerful blueprints that are so rare to find in software development (consider not having to go into the code to understand the design). In this role the patterns can help with the problems of non-understandable designs. By considering the perhaps advanced solutions as self explanatory (they will be once the patterns are learned and understood), the design could be concentrated to the application specific details and leave the patterns used with a simple reference (to a book that will explain it even better). This could be seen as a kind of design by difference (a parallel with inheritance). In this way, the pattern vocabulary will constantly increase and the software engineer’s capability to understand advanced designs will increase.

Besides the documentation capabilities the design patterns carry, they are also very good design pieces that shape robust designs which will not break the first time a change is need

> *The beauty of patterns is that there usefulness will increase exponentially the more developers consciously use them, and the more well known they become.*

References

[Hanseth, Monteiro94]. Ole Hanseth, Eric Monteiro, ‘On what it takes to reuse software’, February 3, 1994

[Sitarama, Eichman]. Murali Sitarama, David Eichman, ‘Inheritance for Software Reuse: The Good, The Bad and The Ugly’

[Aksit, Bergmans]. Mehemet Aksit, Lodewijk Bergmans, ‘Obstacles in Object-Oriented Software Development’, OOPSLA 1992

[Gamma, Helm, Johnson, Vlissides]. Erich Gamma, Richard Helm, Ralph Johnsson, John Vlisside, ‘Design Patterns: Abstraction and Reuse of Object-Oriented Design’

[Gamma, Helm, Johnson, Vlissides94]. Erich Gamma, Richard Helm, Ralph Johnsson, John Vlisside, ‘Design Patterns - Elements of Reusable Object-Oriented Software’, ISBN 0-201-63361-2

[Johnson, Foot]. Ralph E. Johnsson, Brian Foot, ‘Designing Reusable Classes’

[Lajoie, Keller]. Richard Lajoie, Rudolf K. Keller, ‘Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert’

Encapsulation of a frameworks internal interface in C++

Charlie Svahnberg
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration, S-372 25
e-mail: Charlie.Svahnberg@ide.hk-r.se
URL: http://www.pt.hk-r.se/~skalle/

Abstract

This paper introduces a method to enforce encapsulation of framework internal interface written in C++. Usually what framework external interface that are to be used are described in the documentation that comes along with the framework. But no encapsulation of the internal interface is explicitly done. No guarantee exists that the application built using the framework, only uses the external interface, without tampering with the internals of the framework.

1 Introduction

When creating a frameworks, a general behaviour is identified that can be reused in other applications in the same domain. A frameworks consists of classes tightly knitted to each other. The classes in the framework can be both abstract and concrete [Hürsch 94]. Building a framework consists mainly in identifying the interfaces of the abstract classes and how these classes relate to each other, and constructing the concrete classes general enough to be reusable. The general parts of the framework can be implemented in its concrete classes, while the actual behaviour of the abstract classes are unique for each application using the framework, and therefore each application builder have to support the correct concrete classes and hook them onto the abstract ones in the framework. The classes used in the framework to hook onto are called hook classes. The hooking can be done either on to a concrete or an abstract class, that is by instantiation or by subclassing. Although most common the hooking is done by subclassing.

The interface of a class is represented by the available methods operating on the data. In C++ there are means to make sure that the defined interface is the only way to access the internal data of an object. By using keywords like `private`, `public` and `protected` the data members can be protected at different levels from being accessed from outside of the object.

Given this we would like to state that the hook classes of the framework, comprises the interface of the framework, just as the methods in a class presents the interface of that class.

As well as encapsulation on class level, where only a subset of the methods operating on the instantiated object can be reached from the outside, it would be feasible to show a subset of the classes in the frameworks to the outside, that is the application.

One solution to prevent this misuse of the framework is to make the non-hook-classes less reachable by the application implementor. One way to achieve this is simply not to include the non-hook-classes in the distribution of the framework. But this might lead to problems when the hook class needs to know the interface of those classes. The method presented here solves the problem more gracefully, but not totally without trade-offs as we will see

1.1 Overview of this paper

In section 2 we identify some problems concerning frameworks and how they are encapsulated. In section 3, we will describe the student project G^2, we will use it as example further on. Section 4 describes some C++ constructs we are going to use in this paper. In section 5 we present our solution to the problems found in section 3, and in section 6 we discuss and try to foresee some of the problems our solution might introduce. Finally in section 7 and 8, we try to find other similar problems and solutions and try to make a conclusion of our method.

2 Problems with frameworks

2.1 Misuse of framework

The definition of the interface of the framework is done in the documentation, but supplying a framework with proper documentation does not ensure that the intended hook-classes are actually used as hook-classes by the application implementor. The application implementor might instantiate classes from the framework that originally were not intended to be used in such way.

In [Lajoie 94] the possibility to misuse a framework is identified, but not further examined. We believe that the possibility to misuse a framework, due to incomplete encapsulation facilities will result in frameworks being used less frequently than they should. To avoid this scenario we would like to introduce a level of encapsulation in the framework similar to the one supported by most object oriented languages to restrict access to an objects data members.

2.2 Interfaces in a framework

A framework have many different types of interfaces, depending on what level you choose to look at the framework, different types of interfaces will be needed. The interfaces we concern ourselves mostly with in this paper are:

- Each framework supplies an interface to its external clients, framework external interface (FEI) [Deutsch 89].
- Inside the framework, the interface between it's internal clients, framework internal interface (FII) [Deutsch 89].

The problem we try to address in this paper is the lack of encapsulation of the framework internal interface (FII). We try to solve this without breaking the encapsulation of the interfaces of the participating objects. Just as the user of a class need not, and should not, be able to access it's datamembers. The user of a framework, should not be able to subclass or instantiate from classes not belonging to the framwork external interface.

2.3 Black box and White box frameworks

White box frameworks are frameworks which are used by subclassing from the hook classes. To write new subclasses internal knowledge of the framework must be obtained. This means that you need to learn how the framework was constructed before you can learn how to use it [Johnson 88]. What should be noted is that the subclassing need not be done from one single hook class. But there might exist, and also often does, many different hook classes to subclass from to add specific behaviour for the different parts of the framework.

White Box frameworks could be suitable when the framework builder and application builder are the same, No huge mass of new knowledge have to be learnt before building the application. But the real reusability of frameworks comes when different application builders exists, just like class libraries are used by different developers.

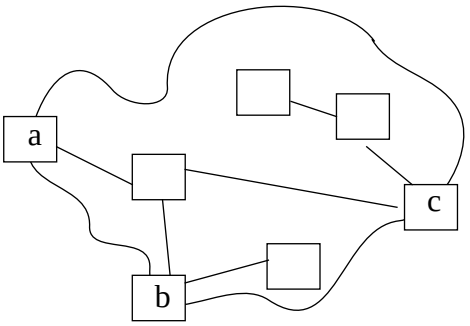


figure 1: Whitebox framework. a,b and c is the external interface of the framework. But the inner parts are visible.

In black box frameworks, the relations between the different part in the framework are defined by the protocol instead of through inheritance [Johnson 88]. This gives that little if any knowledge is needed of the internal structure of the framework. In the long run black box frameworks are more (re)usable.

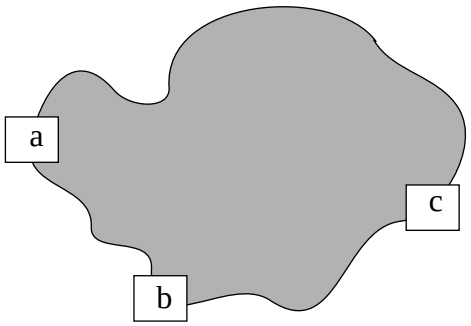


figure 2: Blackbox framework. a,b and c is the external interface of the framework, and the inner parts are not visible

Black box frameworks have a higher need for proper encapsulation of the internal interface than white box frameworks. But white box frameworks, are white just because there is no encapsulation, the internal interface is visible to the user, and because of this the user need to read and understand the documentation of the framework. In reality we believe frameworks to be grey boxed, not purely black box or white box, but a mix of the two.

2.4 Problems due to lack of encapsulation

- Complexity. The framework will become much more complex to learn and use, if not proper encapsulation of the inner parts are done. This is one of the problems with white box frameworks.

- Maintainability. If an application have used the internal interface of the framework, making instances of, or subclassing from non-hook classes. Then when a new version of the framework is released, that version need to be specialised to the applications depending on it's internal structure, or all applications depending on the frameworks internal structure will have to be rewritten. If instead the internals of the framework where hidden from the user, who only were allowed to use the external interface. The new version could be incorporated without mayor problems. In the best case no changes to the application specific code need to made at all.
- Generality, If the development of newer versions of the framework are restricted by how older applications choose to use the internal interface, then generality will be lost, one of the cornerstones of frameworks.
- Reusability. If generality is lost, the possibility to reuse the framework decreases. Since application specific code have found its way into the framework.

The problems mentioned above give us reason to find methods or ways to increase the encapsulation of a frameworks internal interface. We will provide a solution relying heavily on language specific constructs in C++, and therefore our method might unfortunately not be portable to other languages.

3 The G^2 Project

3.1 Introduction

As an example of a framework I will use the results of the G^2 project which was the large team project course for the software engineering students at the University of Karlskrona-Ronneby in the spring of 1993. About 30 students where divided into two groups. Each of these groups were to, independent of each other, develop a framework for a general gateway for telephone-exchange communication within 15 weeks. By implementing a framework the user of our product could connect any types of telephone-exchanges that he/she liked with each other in his/her application.

The main part of the whole project was to come up with a (re)useable design. To see that the framework was usable, an application was built using the framework. Very much care was taken, not to include any application specific knowledge in the framework.

3.2 What is a general gateway

The purpose of a gateway is to store, convert and transport messages between different networks. Normally a totally new gateway need to be constructed from scratch more or less when a new network, with a different protocol is introduced.

By constructing a general gateway, in this case a framework, the core of the gateway is implemented and available for reuse. Only application specific parts need to be (re)constructed. Much less time and money are required to configure your gateway to new environments.

3.3 The object model for G^2

Figure 3 shows the final object model of the general gateway, slightly abbreviated for simplicity. As can be seen all methods and datamembers have been removed. Also the less important classes have been removed. The classes belonging to the framework is displayed, and the empty classes are application specific derivations from the framework. For example in one application an IngresDatabase class can be derived from Database, and a RS235Port could be derived from Port.

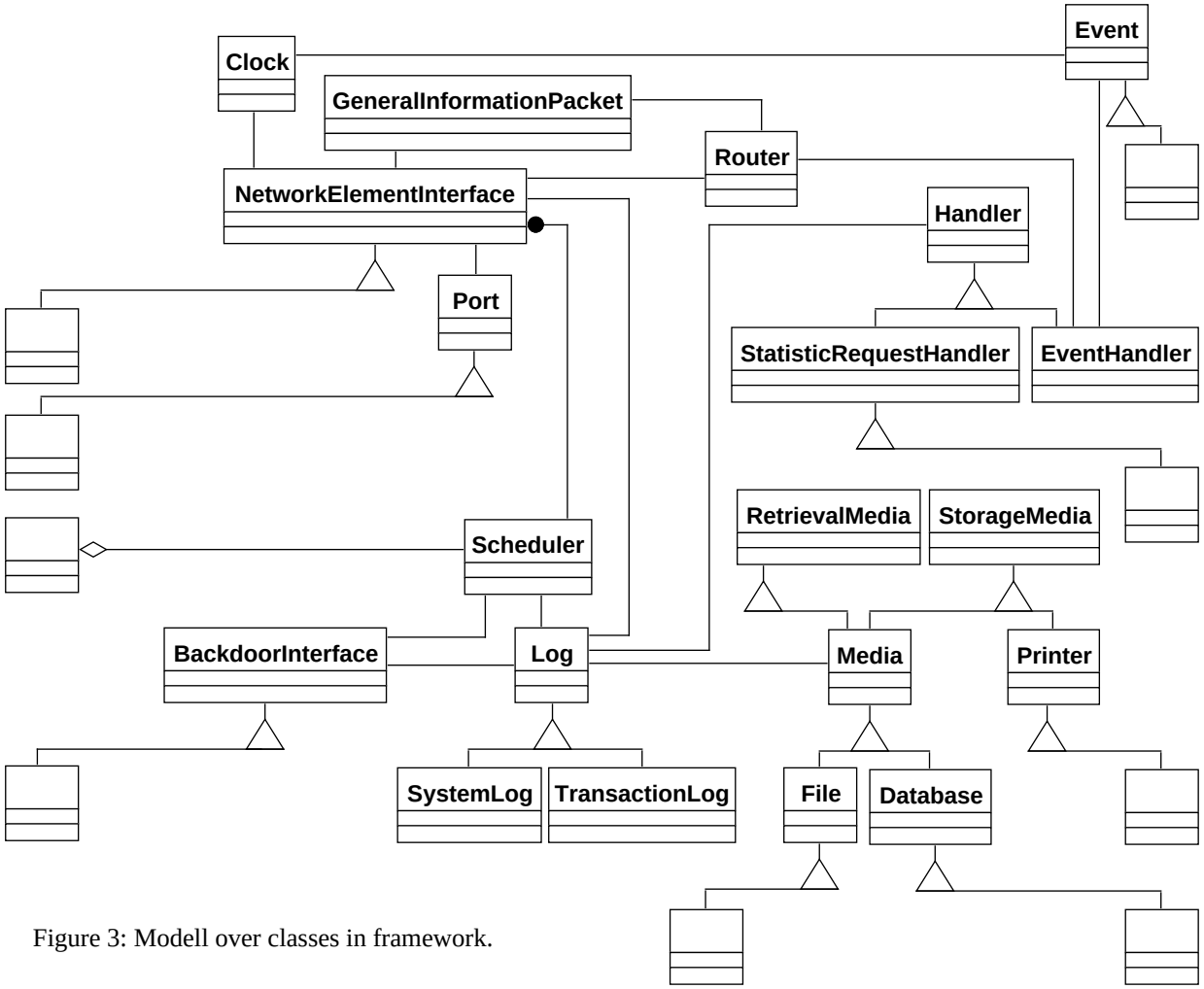


Figure 3: Modell over classes in framework.

3.4 Object Interaction diagram

When designing a system, you should be aware of all interfaces between cooperating objects. You should know what classes that are going to use the methods in the interface of each class. In this paper we are mostly interested in what classes creates and delete objects of each other. In figure 4 we have used the C++ syntax to denote constructors (creation) and destructors (removal), that is `Router` as constructor, and `~Router` as destructor. Using the Router class as example, we want to have a model like the one in figure 4.

The benefits are that you at the design stage get aware of who is going to call the different methods in each class, and also what other classes the current class is depending upon. This make for instance interface changes easier to maintain. In the future it might, if possible, even be feasible to only show a subset of the classes complete interface to the different classes operating on it. Thus encapsulating the interface even more. However we will use this model to identify what classes are allowed to communicate with which other classes.

3.5 The interface of the G^2 framework

According to our definition in the introduction the interface of the framework consists of the hook classes. From the object model over G^2 we can se that its interface consists of the following classes: NetworkElementInterface, Port, Scheduler, BackdoorInterface. Plus the Media, Handler and Event threes.

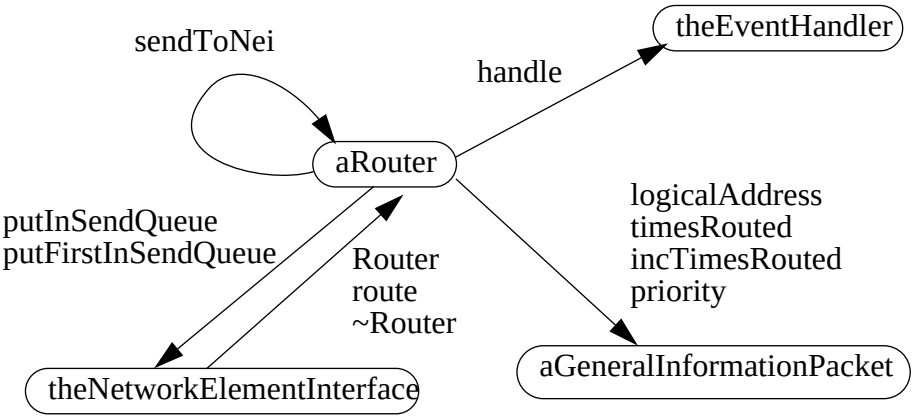


Figure 4: Object interaction diagram for Router

The definition of what subclass to use in the framework, for instance what type of database that is going to be used, where collected to be made in the main program [G^2-1 93b]. The user of the framework, need only to edit the main program to add his/her classes to the application using the framework. The user does not needto make any changes to the classes in the framework.

3.6 Problems identified in the G^2 project

During the G^2 project we identified the possibility to misuse the framework, by instantiating and/or sub-classing from classes within the framework, that were not intended to be used that way in the design of the framework. The framework constructor(s) have to trust the application builder, using the framework, to read and understand the Programmers manual, see [Lajoie 94] for discussion of this problem.

The documentation of the framework were the only mean to give some advice on how to use the product. Although no method existed to prevent our framework from being misused. Therefore it would be feasible to introduce a level of protection from misuse of the framework.

4 C++ concepts and keywords

4.1 Introduction

In the solution to the problem identified in this paper we make extensive use of the encapsulation concept and nested classes in C++, and therefore some background explanation might be needed. Basically encapsulation in C++ are maintained in by four keywords: `private, protect, public` and `friend`.

4.2 Encapsulation in C++

In the class declaration, the visibility of the different methods and datamembers are controlled by inserting the keywords `private, protected, public.` These keywords can also be used in combination with inheritance, but that is outside the scope of this paper.

The keyword `public` is used to grant all users of objects instantiated from the Router-class access to the methods declared within that part. Methods and data in the protected area can only be accessed from the instances of the class itself, it's friends and subclasses. And finally the private area can only be accessed from instantiated object of the `Router` type and it's friend.

```
//Parts removed

class Router
{
   private:
      static EventHandler* theEventHandler;
      //Parts removed

      void sendToNEI(Gip* thisGIP, int toPhysicalAddress);

   public:
      Router();
      void route(Gip* thisGIP);
      ~Router();
};

//Parts removed
```

Figure 5: Declaration of Router

In the class declaration you can also declare what classes and/or methods in certain classes and/or functions that are trusted, and grant them access to private and protected areas as well. This is done by using the `friend` keyword. This concept breaks the encapsulation of the class and should therefore be used with care.

Access type	***Methods in class itself***	***Friends***	***Subclasses to the class***	***Other classes, functions***
public	**X**	**X**	**X**	**X**
protected	**X**	**X**	**X**	-
private	**X**	**X**	-	-

4.3 Nested classes

In C++ you are allowed to nest your class declarations. Inside the declaration of a class another classes declaration can take place. Nested classes can be used to separate the representation of a class from it's interface. See [Coplien 92] on examples of representation.

5 Method to encapsulate a C++ frameworks internal interface

5.1 Encapsulation at different levels

As mentioned in section 2. A framework presents many different interfaces. There are the external interface which the user of the framework uses. The internal interface, that the cooperating classes within the framework makes use of. Plus each individual object have a interface of it's own. Most object oriented languages have some form of encapsulation mechanism that protects the internals of the object from being seen from the outside. When constructing a framework, we would like the internals of the framework to be equally protected. Given here is a solution that encapsulate the internals of the framework, without breaking the encapsulation at object level.

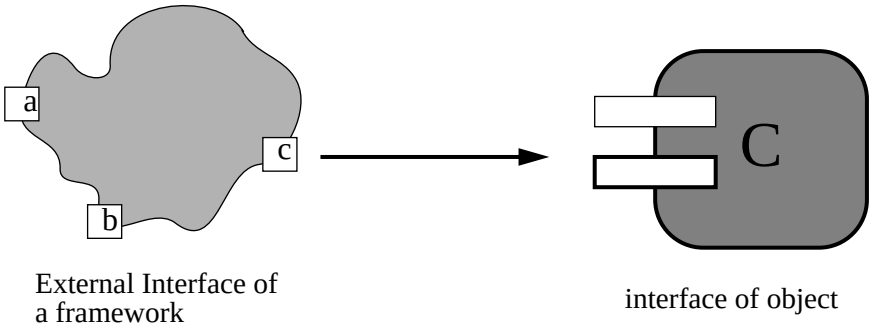


figure 6: two different types of interface within a framework

5.1.1 Encapsulating the internal interface

What we would like to do is restrict the possibility to subclass and instantiate certain classes. Normally inheritance is controlled by the subclass. The subclass choose to inherit from a certain superclass. The superclass do not know anything about the subclasses inheriting from it, if there are anyone at all. We would like to be able to control what classes that are allowed to be subclasses of our superclass. The same applies for instantiation.

In C++ this can be achieved by declaring the constructor and destructor in the private area of the class, then it becomes impossible to create instances directly of that class. Subclassing is also prevented, since when the constructor of the subclass need to traverse the inheritance tree upwards, trying to find its parents constructor. The compiler will check this at compile time. We have now efficiently denied all objects of different type than the class itself from instantiating new objects of that type. It have also become impossible to create any subclasses of our class..

```
//Parts removed

class Router
{
   friend class NetworkElementInterface;

   private:
      static EventHandler* theEventHandler;
      //Parts removed

      void sendToNEI(Gip* thisGIP, int toPhysicalAddress);
      Router();
      ~Router();

      public:
         void route(Gip* thisGIP);
};

//Parts removed
```

figure 7: Declaration of Router, making constructor and destructor private

What we need to do now is look at our Object interaction diagram (see figure 4 previously) and identify what classes that need to communicate with, or rather create and delete instances of, our protected class. We also need to look at the object model (figure 3) to find what subclasses in the framework that inherit

from our class Declaring those classes as friends[1] grants them access to the constructor and destructor, and by that allowing them to inherit and instantiate from that class. See figure 7 for example.

Now, only the intended classes can access our Router-class, we have enforced the encapsulation on one level, the framework internal interface is better encapsulated. However we have totaly broken it on another level, on the object level. One effect of declaring the other classes as friends, is that not only the constructor and destructor becomes visible, but also the datamembers.

5.1.2 Encapsulating the participating objects

On object level we wish to encapsulate the state of the object plus perhaps some special behaviour. One way to achieve this would be to export the state of the object to an external class, that will represent the orignal object, and replace it with a handle to the representing object. All that is left in the original object is the methods that belongs to the objects external interface. We could say that we have almost seperated the state from the behaviour. The representing object will have a state and possibly some behaviour. But the original object will only have a behaviour, using a handle to access it's state. Creating a special representation class will have other benefits and uses as well, some of them identified in [Coplien 92]. The class with it's representation elsewhere can be seen as a pseudo class, that redirects all accesses to it's datamembers to the representing class. We have removed all parts that do not directly concern the interface of the class, and placed them in an, from the outside, unaccessable class.In this way we also remove the reason for encapsulating the inner parts of the class, since there exists no inner parts.

```
//Parts removed

class Router
{
   friend class NetworkElementInterface;

   private:
      class RouterRepresentation
      {
         friend class Router;

         private:
            static EventHandler* theEventHandler;
            //Parts removed

            void sendToNEI(Gip* thisGIP, int toPhysicalAddress);
      };

      RouterRepresentation* myRepresentation;

      Router();
      ~Router();

   public:
      void route(Gip* thisGIP);
};

//Parts removed
```

figure 8: Router with representation

In C++ we would solve it in the following way, if within each class a nested class is created that only holds the memberdata of the class. The representation of the class is kept in a separate class, as described in

1. Sometimes it will suffice to only declare the constructors and destructor of the creating/deleting classes as friends to the class. This will only work if the creation/deleting is done in the constructors/destructros of the creating/deleting class. This is often the case if an aggreagation relationship exists between the classes.

[Coplien 92]. The representation need not be nested within the class it is representing, but we choose to do so, to keep the representation as close as possible to what it is representing. However if declaring the representation outside the class it is representing is more feasible, we can not foresee any problems at this stage in doing so.

In figure 8 we see the Router class with encapsulation of the member data. The classes we previously declared as friends, can see that there is a representation, they can copy the myRepresentation pointer, but they can not use it except as an argument in a method call. It is also impossible to use the pointer to peek into the actual representation of the Router class. One problem is that, in C++, if not specified, among other things, a default constructor, copy constructor and destructor will be generated, and since those generated methods are automatically placed in the public area, it is possible for NetworkElementInterface objects to actually create new objects of RouterRepresentiation. Still worse, NetworkElementInterface objects are allowed to delete the representation, which makes it possible to totally mess up the internal state of the Router

The solution to this is of cause not to rely on the generated constructors and destructor, but to declare those explicit in the private area[1]. This restricts creation/deleting of the Representation to it's friends only, i.e. the class it is representing.

5.2 Accessing internal interface through a hook class

One problem that arises from this solution is that, when subclassing from a hook class, the hook class itself is a friend of the classes it communicates with within the framework, but the subclass is not. Since the subclass is not necessarily written by the framework supplier, it might need to communicate with classes within the framework differently than the hook class. By still putting the interface of all classes in the framework in respective public area. (in section 5.1.1 we only declared the constructor and destructor private), and making the connection to those classes used by the hook classes visible to its subclasses, this problem is solved quite easily.

```
//Parts removed

class NEInterface
{
   protected:
      static Log*                         theSystemLog;
      static Log*                         theTransactionLog;
      static Router*                      theRouter;
      static Clock*                       theClock;
      Port*         myPort;

      NEInterface(int theId, Port* aPort);
      //Parts removed

   public:
      virtual Boolean shutdown() = 0;
      virtual ~NEInterface();
      //Parts removed
};

//Parts removed
```

figure 9: Network Element Interface deklaration

In figure 9 the declaration of the superclass can be seen. Since it is a hook class, and therefore part of the external interface. Preventing inheritance or instantiation is not feasible, that is why the datamembers are

1. If the Orthodox Canonical Class Form, one of the idioms described in [Coplien 92], is used you are required to over write the automatically generated methods.

not placed in a special representation class. In figure 10 an example can be seen of an application dependant subclass. The subclass can freely access the datamembers declared in the superclass, since they are inherited.

```
//Parts removed

class MD110Interface : public NEInterface
{
   //Parts removed
   public:
      virtual Boolean shutdown()
      {
         //Parts removed
         theRouter->route(shutdownGIP);
         //Parts removed
      }
};

//Parts removed
```

Figure 10: Example of subclass of hook class

6 Problems introduced with this solution

Every coin have to sides. The method presented here as a solution to the problems identified in section 2, have some drawbacks itself. We mention and discuss the more serious ones here briefly.

6.1 Flexibility versus Control

One mayor argument against introducing this higher degree of encapsulation is that flexibility is reduced, to much control over what are allowed and not is added. The user of the framework might want to do things not thought of by it's constructors, therefore the user should be allowed to subclass or instantiate from any part in the framework. We do not believe that this is such a big problem. Parallels can be made to being allowed to access data members directly in each object. That could be flexible, in the short term, but what happens when the internal parts are rearranged or replaced. Then freedom becomes a burden. We believe that it is each constructor's responsibility to make sure that his/her class or framework is properly encapsulated on all levels.

6.2 Accessing internal interface through a hook class

Taken a subclass, hooked to the framework. That is, the subclass itself is not part of the framework, but it's superclass is. If that subclass need to instantiate a class directly within the framework, that will not be possible, since that internal class is protected and can not be accessed by non friends, which the subclass most likely is. The superclass might be allowed to create objects of the desired type, and then the subclass can use those instances. But if the superclass do not have access to the desired class then instantiation becomes impossible.

But, the underlying problem is that you want the desired class to behave like a hook class, which it is not defined as. Either the framework designers did not think of it as a hook class, when they should have, or our protection mechanism works properly, and access is denied as it should be.

6.3 Extra overhead

6.3.1 More code

Programmers are lazy, and if a method introduced to them involves to much overhead it will not be used. We believe that the method here is on an acceptable level concerning this. Some extra code have to be written and object interaction diagrams need to be made. Although the diagrams are useful in other aspects as well. But we think that the profit of encapsulating the frameworks internal interface is well worth the cost the extra overhead introduces.

If CASE tools are created in the future supporting the method presented here, the overhead might be delegated to the automatic code generation.

6.3.2 Performance

We do not consider performance to be a problem if our method is used. The extra overhead could possibly mean some extra compilation time. We have introduced an extra redirection when accessing datamembers. But if performance becomes a problem, we are sure that there exists optimization possibilities elsewhere in the product that gives higher result.

Normally we think that optimization should be left until it is needed. If performance becomes a problem, then optimization can be made. However when building a framework you do not know what the performance demands will be in the application using the framework. Therefore performance have a higher priority in frameworks, and reusable components, than normally.

7 Related work

7.1 LayOm

In [Bosch 95], Behavioural Relations are described as a part of the Layered Object Model. The interface the calling object can access from the server object can depend on

- What category the calling object belongs to.
- What state the calling object is in.
- Concurrency, if two concurrent objects tries to call the same object, only one at a time should be allowed to access the methods of the server object.
- Real-time, each message is combined with a time interval that it must be executed within, if that interval have expired access should be denied.

The solution described in [Bosch 95] gives a more general solution to the underlying problem of steering what objects that are allowed to call each other, that this paper have identified as a problem especially within frameworks.

Behavioural relations would solve the problem of misuse of frameworks in a nicer way than described here, but for the moment that feature is not available in C++, and since C++ is commonly used, a solution oriented towards C++ is needed.

8 Conclusion

We will describe how we think the different problems identified in section 3 have been fulfilled if our method is used.

- Complexity. By encapsulating the frameworks internal interface, and in that way making it less white. The complexity of learning and using the framework is reduced. You no longer need to know everything about how the framework operates internally, since you are not allowed to make use of it's internal parts anyway.
- Robustness. It becomes much more unlikely that the user of the framework will mess up the behaviour of the framework by calling wrong methods, instantiate wrong classes, or subclass from wrong classes.
- Maintainability. With code that supports encapsulation of the internal interface of the framework, it becomes less likely that an application builder will have to make extensive changes in his application when a new version of the framework is released. In the best case, no changes are need to be done at all, a recompilation would suffice. This is the most probable situation if internal bugs where fixed in the framework, or if it have been optimized.
- Generality. The builder of the framework, knows that the only parts that can be used by the application builders is the external interface of the framework. When changing the framework, no consideration have to be taken regarding the different parts of the framework except for the external interface. There is no risk that there exists applications depending on the internal structure of the framework, and that these dependencies are thought of when updating the framework.
- Reusability. Since the risk that specialisation, to support an application depending on the internal interface of the framework, have been made in the new version of the framework is minimized. It can be reused in application's that do not work exactly as the one the framework supports.

The idea that a framework can support a special set of applications might seem strange, but it is very easy to, without knowing it, make decisions when constructing, or updating, a framework that are affected by the applications using it.

One major drawback is that some flexibility is lost, possibilities not thought of by the framework-designers are made impossible to use. But the problem here is the same as in encapsulation of classes or objects. Flexibility versus Control. To much flexibility is not good, the possibility for reuse will normally decrease. To much control will also restrict the possibility for reuse.

Acknowledgements

Mikael Mattsson, for commenting and for pointing out some relevant resources.

Frederik Bonte, stimulating discussions.

Jan Bosch, reading and commenting on the content of this paper.

Magnus Broberg, for mentioning and discussing the pseudo class concept.

References

[Coplien 92] James O. Coplien. Advanced C++ programming styles and idioms. Addison Wesley 1992.

[Stroustrup 94] Bjarne Stroustrup. The design and evolution of C++. Addison Wesley 1994.

[Lajoie 94] Richard Lajoie, Rudolf K. Keller. Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert. 1994.

[G^2-2 93] PT91, Software Engineering program. Sales Guide for G^2-2. G^2-2 1993.

[Bengtsson 96] PO Bengtsson. Object Oriented Frameworks Definition. AOOC 1996.

[Hünsch 94] Walter L. Hürsch. Should superclasses be abstract? ECOOP 1994.

[G^2-1 93a] PT91, Software Engineering program, General gateway design. G^2-1 1993.

[G^2-1 93b] PT91, Software Engineering program, Programmers manual v1.0. G^2-1 1993.

[Johnson 88] Ralph E Johnson, Brian Foote. Designing reusable classes. JOOP 1988.

[Bosch 95] Jan Bosch. Layered Object Model, investigating paradigm extensibility. Ph D thesis 1995.

[Deutsch 89] L. Peter Deutsch. Design reuse and frameworks in the smalltalk-80 system. Software Reusability (Ted J Biggerstaff), Addison-Wesley 1989.

Does C++ support reuse?

Ulf Kristiansson
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: mpt95ukr@pt.hk-r.se
www: http://www.pt.hk-r.se/~mpt95ukr/

Abstract
Reuse of software have been addressed in many reports, this report will be different in that way that it is addressing how software can be reused and diffrent concepts in C++ that makes reuse meaningful. It will also take up how the diffrent kinds of reuse are realized in C++ and what to think of when writing programs that shall be reused. The purpose whit the article is to give a broader and deeper understanding of how to reuse in C++.

Keywords
reuse, C++

1 Introduction

Software can be reused in many different ways. As a purchaser it is possible to reuse software by buying it of the shelf or by extending the current system. As a developer it is possible to reuse by design or by reusing parts of the program code. This paper will not take up the reuse by design aspects or how to document the programs, instead it will be concentrated on how to reuse program code in the programming language C++ and how to avoid those problems that can occur when reusing.

First this report will take up different general aspects of how things can be reused. It will be followed by a presentation of diffrent programming concepts in C++. After that there will be a description of how the different kinds of reuse can be fit in with C++. To end with there will be a conclusion containing what to think about when writing programs that can be or will be reused.

2 What is reuse

Reuse is when one thing are used many times. The thing can be used to different things but it is still the same thing that are used. For example a newspaper, it can be read and it can be put under one of the legs of a table to stabilize it. If reuse in software engineering are looked at from this angle it should mean that the same executable program is used to many different things. This is not the case. In software engineering when talking about reuse, or reengineering, it is to use the same component in many programs. The components can be large, almost a whole system, and they can be small, for example a function that change a value from Fahrenheit to Celsius. To make this possible is it important that the components that shall be reused have well defined interfaces. This is important of two reasons, to know what the component can do and how to fit it together with the other components.

When reusing components it can be good to modify these so they better fit the purpose. When using object-oriented languages this can be done by inheriting objects[1]. Inherit an object is to get a new object that are exactly the same as the inherited object and to this new object new things can be added and the old can be

changed but not removed. An other way to modify a component is to make a copy of it and change the copy.

Components that shall be reused can be done so by reference or by replication (copying). Replication of the component that shall be reused is not always a good way to reuse in. Changes to the original does not reflect on the replica and vice versa. If there for example is something wrong with the original and it is corrected, the user of a replica will still have the problem. This will not happen when reusing by reference, since the same component is used as both the original program and the new program.

The component that shall be reused can be done so in two different ways, as-is or by-adoption. As-is is to use it straight of and by-adoption is when some extensions or/and some modification are made to the component.

In [Vliet 91] "Succeeding with Objects" they take up three different types of reuse. These are the black box reuse, glass box reuse and white box reuse.

- Black box reuse

  Black box reuse are when only the outside of the box can be seen. The outside of the box can also be called the interface, what to do not how to do it, of the box. When using black box reuse it is not possible to change the things that are reused. Black box reusing is of the reusing type as-is and it can be done with either by reference or by replication. Due to the black box well defined interface it is easy to fit it together with other objects. The quality of a black box is provided by the builder of the box. The user of a black box only have to test their use of the black box.

- Glass box reuse

  Glass box reuse are when both the outside and the inside of the box can be seen. The inside of the box are also known as the implementation of the interface. It is not possible to change the things that are reused. The difference from black boxes is that it is possible to see the inside of the glass box and understand its internal structure. To see the internal representation can be good for learning how to build new boxes. Sometimes it can be critical to know how the box works to reuse it. It can also be negative to see how the data are represented and how the algorithms are implemented. This is because other parts of the system may then rely on how the box works. If the box's internal structures or algorithms changes, it is probably so that the system will not work as it was supposed.

- White box reuse

  White box reuse are when both the outside and the inside of the box can be seen. It is also possible to change the inside as well as the outside of the box. Because of the accessibility in white box reuse my personal opinion is that wire box reuse is a better word. This is based on that a wire box are better reflecting the fact that the inside of the box can be accessed, if the box is white it is hard to picture that so is the case.

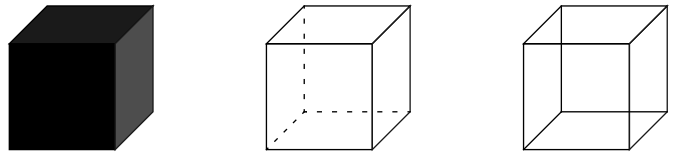


Figure 1. Different kinds of reuse.

General the boxes can be seen as a box with different attributes. A black box can be seen as an closed coloured box, a glass box as a closed transparent box and a white box as a open transparent box.

1.Objects refers also to classes when it is not concerning implementation aspects in C++.

This three concepts are only mental pictures of how it is possible to reuse. In most languages it is when reusing the pieces the kind of reuse is chosen, not when the pieces is constructed.

3 C++

There are some basic concepts that shall be fulfilled in an object-oriented languages. These are objects, encapsulation, inheritance and polymorphism. In this chapter it will be explained how these concepts works in C++.

An important thing about C++ is that it does not differ classes from types. An object of class A will also be of the type A. An other thing is that C++ is build upon method calling instead of message passing as some other languages are, smalltalk for example.

3.1 Objects

In C++ the objects are created during runtime. This is done by creating an instance of a class. Classes in C++ are templates of how the objects shall look like. A class contains description of which data an object of that type contains and which methods there exist to access that data.

3.2 Encapsulation

There are three levels of encapsulation C++. It is possible to declare the methods and data in a class as Private, Protected and Public. The methods and data that are declared as Private can only be accessed be other methods within the object and by methods that are declared as friends to the class. Declaring methods and data as Protected will make them function as they where declared Private. How the method and data can be accessed after the class is inherited will however differ between Private and Protected, how it differs will be explained later. Methods and data that are declared as Public can be accessed by the whole system.

3.3 Inheritance

Inheritance is common known as a a big help in object oriented languages to provide reusability. Inheritance in C++ differs from other languages because it has three levels of encapsulation of the methods and data and two types of inheritance. This leads to the possibility to have a method or data member to be inherited in six different ways.

The two types of inheritance are Public and Private. When inheriting a base class as Public the Public methods and data of the base class[2] will still be Public in the derived class[3]. The Protected methods and data will also stay Protected and can be accessed by all methods in the derived class. The methods and data declared as Private in the base class will not be accessible in the derived class. These methods can only be accessed through the base class methods that are visible in the derived class, the protected and public methods.

When inheriting as Private the Protected methods and data are treated as if they where declared as Private, therefore they can not be accessed directly in the derived class. The Public methods and data will be treated

2. Sometimes referred to as the super lass.

3. Sometimes referred to as the sub class.

as they are declared as Private in the derived class. Therefore they can not be accessed in classes that derives the derived class.

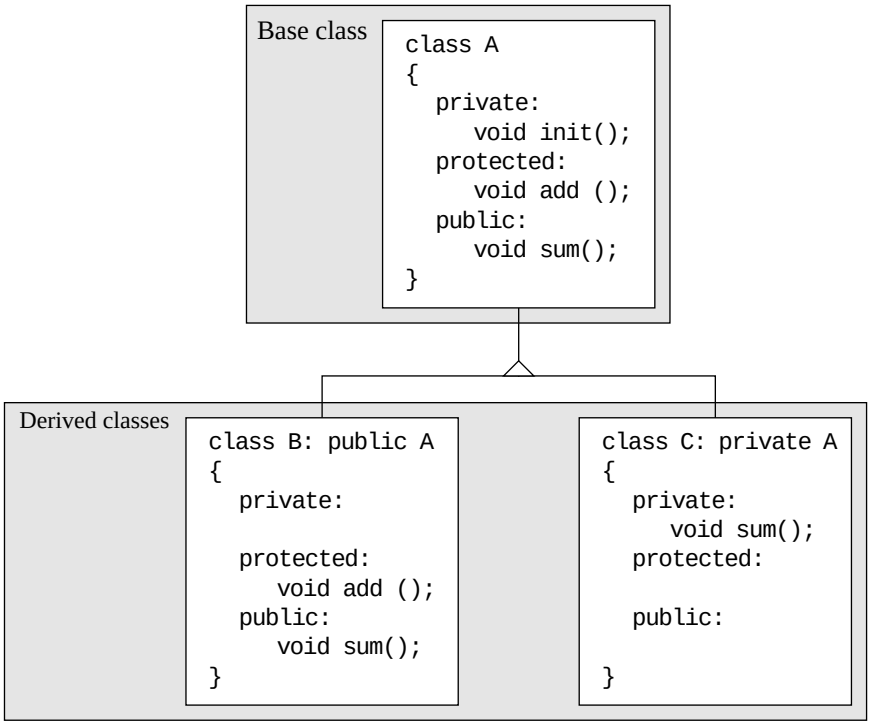


Figure 2. Public and private inheritance.

This can be used to restrict access in deriving classes. But methods and data can be restored to there level of encapsulation by declaring them as the level of encapsulation in the base class and specifying that the base class that shall be used.

```
class C: private A
{
  private:
     void sum();
  protected:
     void A::add()
  public:

}
```

Figure 3. Restoring the level of encapsulation.

3.3.1 Multiple inheritance

In C++ it is possible to inherit from several base classes. This makes it possible to inherit the behaviour and characteristic from two different classes into a new class. The declaration is made in the following way:

```
class Derived: public BaseA, public BaseB
```

When a method or data are declared in several base classes the method or data from the base class first found in the list, read from left to right, is chosen. If a method from not the first base class found shall be used the method must be overloaded. The overloaded function can then specify which method that shall be used. Consider the example in Figure 4. on page 5. When calling the sum method in class C the implemen-

tation of class A will be called. When calling the sum method in class D first D's sum method is called and then it is calling B's sum method.

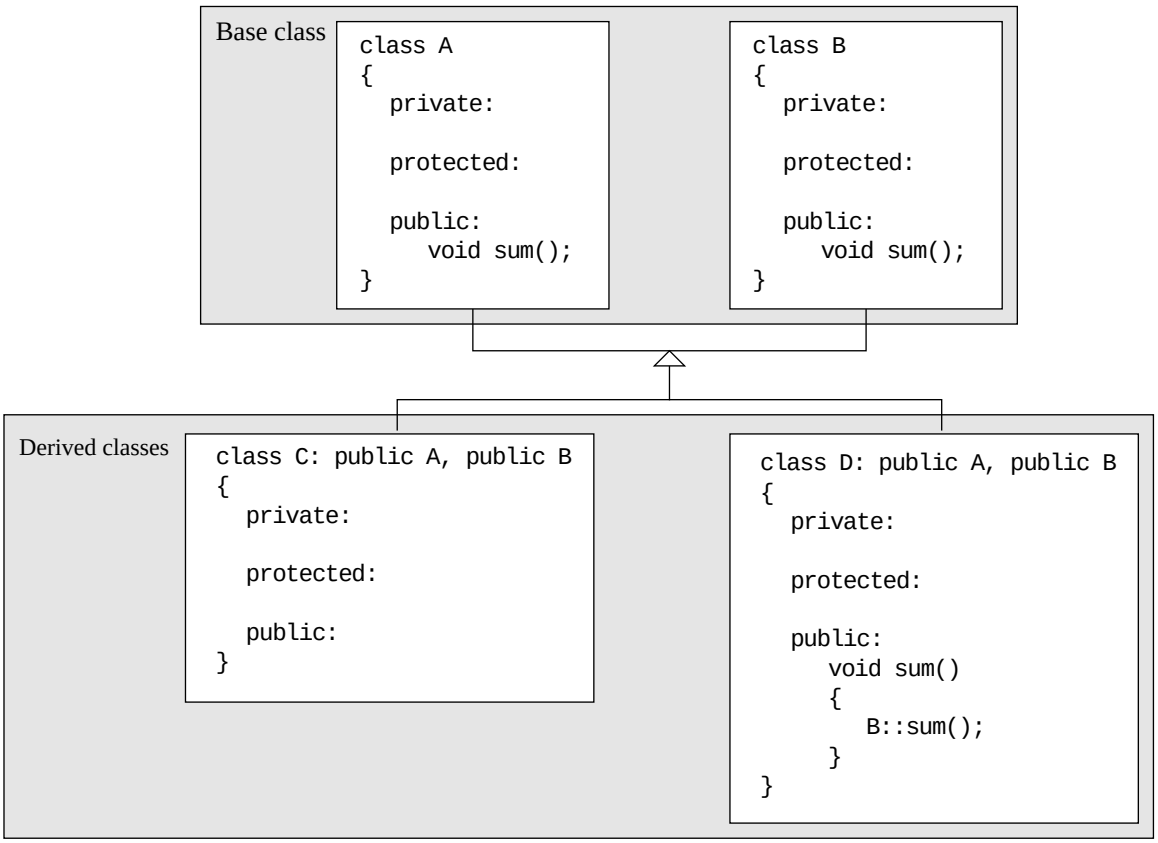


Figure 4. How to access derived methods.

When accessing data that are declared in several base classes it is instantiated several instances of that data. Therefore it is necessary to specify from which base class the data shall be accessed from. If the base class is not specified the compiler will complain about it.

If the base classes, A and B, are derived from the same base class, E, it will be problems when accessing data declared in class E from either class C or D. This is because both class A and B are initializing one instance each of class E. This can be avoided in C++ by declare the inheritance of class E as virtual. So by declaring both class A and B as virtual public E instead of public E it is guaranteed that only one instance of class E will be instantiated.

3.4 Polymorphism

Polymorphism is when there exist one interface and several methods. C++ supports two types of polymorphism. These are polymorphism at compile time and polymorphism at run time. Polymorphism at compile time is accomplished by overloaded methods and polymorphism during run time with virtual methods and inheriting.

3.4.1 Overloaded methods

This means that methods can have the same name but perform different actions. Which method that shall be executed are decided based on the parameters sent to it. This means that it is possible to have a set of functions called add which adds two numbers. Depending of the type of the data sent to the method the 'right' method is chosen.

3.4.2 Virtual methods

In C++ it is possible to have a pointer to an object. This pointer is of certain type. The pointer can refer to objects of derived types. When accessing a method the method in the pointers class is chosen. This is not always what is wanted, so to be able to access method that are overloaded in the derived classes through a pointer to a base class the method in the base class must be declared as virtual. The derived classes does not have to have the method. If the class pointed to do not have the method, it is search for the method bottom up in the hierarchy.

3.4.3 Pure virtual methods and abstract classes

It is possible to have methods in the base class that are not implemented. In this way the derived classes are forced to have an implementation of the method. This is done by assigning the virtual methods the value zero.

```
virtual int add(int, int) = 0;
```

These methods are called pure virtual methods. If a class contains at least one pure virtual method that class is said to abstract. This is because the class is missing a description for at least one method and can therefore not be instantiated. Although it is not possible to create objects of an abstract class, it is possible to have pointer of that type.

4 Reuse in C++

Reuse is not anything that happens when using a specific programming language, it is a matter about mentality. To be able to reuse components it is important that they are described in an easy and understandable way. C++ does not have any support for documenting the classes in the language, so this has to be made separately.

4.1 Reuse by reference

Since C++ needs to be compiled it does not support any real reuse by reference. The classes are usually written in two files each, one file with the description (the .h file) and one file with the implementation (the .cc file). In this way it is possible to include a class when writing a new class or program that already have been used. The .h file are included in the file using the class and when compiling the .cc file and all files it includes has to be included. This is very complicated and time-consuming. Since it is possible to read both the description and the implementation of the classes, the natural thing to believe is that this would be glass box reuse. But since there are no restriction in C++ against changing these classes it is white box reuse. If the program is developed on a platform that supports access rights on files then these files could be write protected and thereby it would be glass box reuse.

There is an solution to this problem that the inside of the program are readable. This is to compile the classes that belongs together into a library. When using the library, the .h file are included when the class are used and the whole library when compiling. Often when libraries are made these are made big since they contains several classes that does different things in the same are. A example is libraries that contains collection classes often have lists, queues and stacks, so when using the list classes the other classes are also included. This problem are today a minor problem because today’s compilers are optimizing the code in such a way that parts that are not used in the program will not be included in the executable program. Class libraries are almost black boxes, since the only thing that can be seen is the methods and data of the classes. How the classes are implemented can not been seen.

4.2 Reuse by replication

In general when people are reusing in C++ it is done by replication. A class or method just in a previous written program are being copied into the new program. Often the classes are not exactly what was needed so instead of inheriting the class it is altered. This is clearly white box reuse.

Class libraries can also be replicated but this is not very meaningful since it is better to have all libraries collected at one place so that they are easy to find when needing them.

4.3 Reuse by adoption

The best way to reuse by adoption is to inherit the class that needs to be modified. In the new derived class the methods that needs to be altered are overloaded. When making inheriting hierarchies it is important to think about how they are supposed to be used.

4.3.1 Frameworks

A framework is in a broad general way a skeleton or foundation on which the program is build. The skeleton is almost always made for only one type of system, for example resource booking systems.

The skeleton is made of class hierarchies that are connected together by a basic system. The class hierarchies in a framework are mostly made of abstract classes. These classes must then be inherited when adopting the framework into a program. An important diffrent from when reusing other components, like class libraries, is that it is the framework that calls the methods in the derived classes. Therefore almost every method that can be inherit from the framework is virtual.

Frameworks are typical reuse as glass box. This must be this way because the framework are often made of complex structures that must be known to be able to write the inherited classes in the right way. They should be black box reuse sins only the behaviour of the framework should be known. It will also be needed some information about the classes that shall be created and the classes that they will inherit from.

5 Conclusions

There are several different ways to reuse in. The main thing is to get convinced that it is a positive thing to reuse components. Then it is god to decide in which way things shall be reused. Shall things be but together in a class libraries or shall the classes just be collected in directories.

When the way of reusing is chosen a standard for how this shall be done and how the components shall be documented. The documentation is a very important part that often are neglected. Looking at the class libraries that exist today for example. They are poorly documented if there exist any documentation.

When writing class hierarchies and there is a chance, risk, for multiple inheritance with two or more classes that comes from the same base class I think it is important to declare the inheritance as virtual to avoid problems with data members instantiated more than once.

Acknowledgement

I would just take the opportunity to thank Magnus C. Ohlsson and Magnus Östvall for there help with the english grammar and spelling, but for that sake if any grammatical error or misspelling is found it is me that should be blame, maybe the spelling checker, not them.

References

[Hanseth 94] On what it takes to reuse software
Ole Hanseth (NR), Eric Monteiro (UNIT)

[Sitaraman] Inheritance for Software Reuse: The Good, The Bad and The Ugly
Murali Sitaraman and David Eichmann

[Goldberg 95] Succeeding with Objects
Adele Goldberg and Kenneth S. Rubin
ISBN 0-201-62878-3

[Vliet 91] Software Engineering, principles and practice
Hans van Vliet
ISBN 0-471-93611-1

[Lajoie 94] Design and Reuse in Object-Oriented Frameworks:
Patterns, Contracts and Motif in Concert
Richard Lajoie and Rudolf K. Keller

[Micallef 88] Encapsulation, Reusability and Extensibility in
Object-Oriented Programming Languages
Josephine Micallef

Well Documented Classes

Riquard van der Vliet
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: mpt95rvv@pt.hk-r.se
www: http://www.pt.hk-r.se/~mpt95rvv

Abstract

The object oriented paradigm has several advantages comparing to other non-object oriented paradigms. Inheritance is one of the features of the object oriented paradigm promoting to build reusable components. These components can be viewed as black boxes, but using the inheritance feature the programmer has to understand the complete inherited component, otherwise he will inherit some uncertainty in his own created black box. Therefore the documentation of a component is an important part writing reusable software. Since even compiler-builders have problems to document their libraries sufficient, this paper will introduce a document structure reducing the problems of writing good documentation for components in class-based programming languages.

Keywords: Reuse, Reusability, Documentation.

1 Introduction

Object oriented software is all about objects. An object is a ‘black box’ which receives and send messages. In object oriented programming, code and data are merged into a single indivisible thing, an object. A primary rule of object-oriented programming is this: as the ‘user’ of an object, you should never need to look inside the box. The interface to the object is defined by messages. Everything an object can do is represented by its message.

One feature of object oriented programming is inheritance. Inheritance provides a natural classification for kinds of objects and allows for the commonality of objects to be explicitly taken advantage of in modelling and constructing object systems. Natural means we use concepts,classification, and generalization to understand and deal with the complexities of the real world.

Inheritance promotes reuse. You don’t have to start from scratch when you write a new program. You can reuse an existing repetoire of classes. Although reuse sounds good there are problems with it. A lot of managers are still waiting for a day where the systems will be assembled like lego. This paper will identify several problems with reuse and specify the document part of reusable classes particularly.

This paper is further organized as follows. In chapter 2 we will give the definitions involved with reuse. Chapter 3 will discuss reuse in the real world. Chapter 4 gives an explanation about the importance of documentation. Our implementation of the document part for reusable classes has been put in Chapter 5. Chapter 6 will end-up with a conclusion.

2 Definitions

Software Reuse is in some ways a strange subject. Practically everybody is practising it in some way or other. The general opinion is that reuse is good and that one should try to incorporate reuse of software in one development process where possible. The basic motivation is that you can use prefabricated parts will lead to higher productivity and thus saving costs.
The following are definitions of some of the basic terms used by researchers and practitioners in the reuse

field. There is no single accepted definition for most of these terms; terminology in the field is still evolving. These definition are from [Marciniak].

1. **Reuse**--the use of an existing software component in a new context, either elsewhere in the same system or in another system.
2. **Reusability**--the extent to which a software component is able to reused. Conformance to certain design and coding standards increases a component's reusability.
3. **Reusable software component**--a software entity intended for reuse; may be design, code, or other product of software development process. Reusable software components are sometimes called "software assets".
4. **Reuser**--an individual or organisation that reuses a reusable software component
5. **Portability**--the extent to which a software component originally developed on one computer and operating system. A component's reusability potential is greater if it is easily portable.
6. **Domain**--a class of related software applications. The potential for reuse is generally greater within a single domain.
7. **Library**--a collection of reusable software components together with the procedures and support functions required to provide the components to the user.
8. **Retrieval system**--automated tool that support classification and retrieval of reusable software components, also called a "repository" or "reuse library system".

3 Reuse and Reality

Reuse of software is considered as an important quality criteria. Good programmers shouldn't invent the wheel twice or more. Instead of inventing they have to search in a library to find the right component. Todays managers are still waiting for the day that the system can be assembled using modular components from the library. This sounds great but nobody manages to reuse software sufficient, why? What are the problems and are the advantages really useful. This chapter will discuss some problems of software reuse and gives also the advantages of it.

3.1 Reuse problems

Software reuse is hard to work out sufficiently. We have to know that there exist a lot of different shapes in software reuse. Some progress has been made regarding to application programming interfaces(API's) of networks, operating systems, user interfaces, and database management systems. Although all these progress in software reuse we still can define the following problems:

1. The costs to set up the reusable software process are expensive. We could be thinking about the investments of tools helping the programmer to find the right component in a library.
2. It is difficult to estimate what the effort is of reusable software. How do we know the usage of the reusable class in the (near) future.
3. After a while, motivated and well-educated programmers are disappointed about the reusability of the expensive libraries.
4. Programmers are creative people. They find it more fun writing their own code. Writing the code by yourself is a sign of craftsmanship.
5. There is no education at universities how to reuse software. Students learn how to program, but they haven't got the education how to find the right software component in a library.

6. The documentation of a reusable class is poor.

This paper is only focusing on the last problem. In chapter 4 we will discuss the importance of documentation.

3.2 Reuse advantages.

The major and also the most obvious benefit of software reuse is improved productivity, implying cost savings. The improve productivity isn't only gained in code development but also in analysis, design and testing phases. The cost savings made in the software process allow the businesses to price their product attractively, resulting in a more competitive advantage and market share, and to higher profits. Other examples of expected benefits are [marciniak]:

1. Software components are used in systems, delivered and subjected to day-today use, they will evolve a highly reliable state. Thus reuse can lead to improved quality, reliability and performance(improving and optimizing the components).
2. Reuse typically results in user-perceived uniformity among systems; for example, reusing the user interface software from a previous product when developing its replacement means that users will be able to transition to the new system much more easily.
3. Reuse supports interoperability, the ability of two or more systems to work together. Both systems can use the same software components to implement key interfaces, assuring that these interfaces are implemented consistently
4. Reuse lead to greater standardization. System architectures will become more uniform, algorithms will become more consistent, and development processes will converge. Moving to more standard practice allows software engineers to become productive more quickly on a new product and provides a foundation for more effective quality management disciplines.
5. Reuse supports rapid prototyping, putting together quick operational models of systems, typically to get customer or use feedback on the capability. A library of reusable components provides an extremely effective basis for quickly building application prototypes.

4 The importance of documentation

If systems can be assembled then the programmer/reuser should use his time to find the right component in the library instead of programming the routines. Once, the programmer found a component he/she should understand the total component because in principal it is possible that a programmer defines a child class and this class uses a routine of the parent class i.e. overloading, without considering that the child class has a specific properties which makes the parent routine unsuitable.

In best cases the error can be found at compiletime but if not, a runtime error occurs plus the fact that the programmer build in his system some uncertainties in his own 'black box'. To conclude any programmer would reuse a routine which isn't clear enough for him.

The next question to solve is what do we need to clarify a software component for a programmer as fast as possible? At least we need good documentation, it isn't user-friendly to put all the information in the programming language. The remainder of this chapter we focus on the documentation of the library classes and showing several demerits.

4.1 Comparison between library documentation.

Almost every software developer or manager is aware of the importance of documentation. A lot of people proclaims ‘we have to document, we have to document we have.....’but if you ask for the right way the answer will be ‘we have to document’. Good software documentation is scarce even compiler-builders can’t document their libraries sufficiently, for example:

A programmer wants to develop a simple stack providing such operations as push, pop, print stack, number of elements, etc.
The programmer starts the development by a natural action; he looks in the manual for a class similar to “stack” in order to see which methods are provided to work with it. After some browsing he discovers that where he was looking for was actually named “lifoQueue”. He looked at the documentation of the class “lifoQueue” from every angle and found a single method called “insert”, which was the push method. A bit puzzled (The programmer expected to find all the methods related to the stack management in the class) the programmer had to look for the pop and top methods. In class Queue (the grand father of lifoQueue) the programmer found the “remove” (pop) method, the top method does not exist and to access the top element the programmer have to “make” a position in the stack and access it dereferencing the pointer.

In this chapter we will look at the documentation of the following four class libraries:

- LEDA (Library of Efficient Data types and Algorithms).
- NIHCL (National Institute of Health Class Library).
- The Eiffel base object-oriented component libraries.
- The OSE-libraries (OTCLIB OUXLIB and OTKLIB).

At the end of this chapter we will show the drawbacks of every library.

4.1.1 LEDA Library of Efficient Data types and Algorithms

In the fall of 1988 Naher et all. started a project called LEDA to build a small, but growing library of data types and algorithms in a form which allows them to be used by non-experts. LEDA is a library of efficient data types and algorithms. The implementation language of LEDA is C++. For a detailed description of LEDA we refer to [Naher].

In general the specification of a LEDA data type consist of four parts:

1. A definition part defining the objects comprising the data type using standard mathematical concepts and notations.
2. A description of how to create an object of the data type.
3. A definition of the operation available on the objects of the data type. For each operation the description consist of two parts:1) The interface of the operation is defined using the C++ function declaration syntax. 2) The effect of the operation is defined. Often the arguments have to fulfil certain preconditions.
4. Information about the implementation. This part lists the data structures used to implement the data type and gives the time bounds for the operations and the space requirements.

Sometimes there is also a fifth part showing an example.

4.1.2 NIHCL National Institute of Health Class Library

NICHL (pronounced either N-I-C-H-L or “nickel”) implements abstract data types that have been designed to simplify object-oriented programming using C++. It contains generally useful data types, such as String, Date, and Time, and it provides a set of classes similar to the Smalltalk 80 collection classes including indexed arrays, singly linked lists, hash tables and associative arrays.

NIHCL is documented in [Garlen], which is a book in data abstraction and object oriented programming in C++ within whose framework the class library is described. Therefore the documentation has not the form of a reference manual, but it provides a short description of the different classes with some examples.

This approach has the drawback that when you are developing code and you are already have some knowledge of the library you would probably use more profitably a real reference manual.

4.1.3 The Eiffel base object-oriented component library

The Eiffel base libraries covers several hundred reusable classes grouped into clusters. The clusters are themselves gathered into five main libraries: kernel, data structure, iteration, lex, parse. The Eiffel base libraries are described in [Meyer].

The basic idea of documentation at the Eiffel libraries, is that they avoid treating software and it documentation as separate products. Instead, all the information will appear in the classes implying that the documentation is kept up to date. Eiffel bases the documentation of each class on the flat-short form of the class. The flat-short form is the abstract interface, obtained by flattening and then shortening the class. Flattening means[Meyer] creating a version of the class including the methods inherited from parents as well as the immediate features. The flattened form should take renaming and redeclarations into account., and reconstruct full pre- and postconditions. Shortening means removing all information that is not relevant to client authors. In the short form the method headers, the header comment, the pre and post conditions and invariants will be retained. An example of a short form is given below:

CLASS Traversable

Traversable is an heir of Container and retains all its exported features. This interface only shows the features introduced or redeclared in Traversable.

Indexing (index for finding the class).
description: “structures for which there exists a traversable policy that will visit every element exactly once.
names: traversable,traversing
access: cursor
contents: generic

Deferred class interface
Traversable[G]

Feature -- Access
item: G -- Item at current position
require not_off: not off

Feature -- Status report
off:Boolean -- Is there no current item?

Feature -- Cursor movement
start -- Move to first position if any

End

The documentation of the Eiffel library does not contain any example.

4.1.4 The OSE-libraries

OSE is a collection of programming tools and class libraries for C++, which have been developed with the aim of improving programmer productivity. Increased productivity is achieved through the ability to reuse software components. More information about OSE has been written in [OSE].

OSE contains the following three C++ class libraries:

1. OTCLIB - A library of generic components, including support for error handling, error message logging, error recovery, program debugging, memory management, resource management, generic collections, text manipulation, operating system interfacing.
2. OUXLIB - A library of components which primary extends classes in the OTCLIB library to support features specific to the UNIX operating system
3. OTKLIB - A library of components which builds on the OTCLIB and OUXLIB libraries to allow integration of the TCL/TK library into applications using the event driven systems framework provided by the OTCLIB.

OSE provides some documentation extraction tools allowing documentation about C++ classes to be embedded within C++ header files as C++ comments. OSE argues that the close link between the documentation and code helps to ensure that documentation is kept up to date. The structure they use to describe their classes is as follows:

NAME
Classname - one line description of the class

SYNOPSIS
The structure of the class in pure C++ code i.e. the header file.

CLASS TYPE
Description if the class is a concrete or an abstract class

DESCRIPTION
A description about the class, when to use it. An example has included into the description part

INITIALISATION
The description of how to create an object of this class.

DESTRUCTION
The description of the destructor of the class.

METHODS
The description of the methods in the class.

SEE ALSO
Reference to related classes.

LIBRARY
The name of the library where you can find the class

AUTHOR
The name of the Author.

4.1.5 Demerits in the documentation of the different libraries

At the previous four sub-chapters we gave a short overview of the documentation at a specific library. The documentation of the LEDA library appears to be linear and concise it isn't difficult to read but in most cases it doesn't contain examples of the way the classes have to be used.
The NIHCL is documented in [Garlen]. The documentation doesn't have the form of a reference manual, it is brought as a reading book whereby a short description of the different classes are given plus some examples. This approach has the drawback that when you are developing code and you are already have some knowledge of the library you would probably use more profitably a real reference manual
The Eiffel libraries are documented in [Meyer]. In this book some chapters discuss the documentation problem and gives some recommendation, but one drawback of the Eiffel documentation is that it doesn't include examples.
The documentation of the OSE-libraries seems to appear structured, but through the documentation you will find some inconsistency for examples the header 'initialisation' has sometimes a different name called 'constructor'. In the next chapter we try to reduce the remarkable drawbacks and give our own implementation of class documentation.

5 Our implementation

At every library, evaluated in the previous chapter, there exists at least one drawback. We have the opinion that the documentation should be readable, intuitive and complete. With readable we mean that there exists a structured way to describe the classes. Intuitive means that the reader of the class documentation knows what he is reading. A complete documentation about one class should contain all information kept by the programming code.

[Meyer] says "We should stop to see the software and the documentation as separate products". We fully agree, since separating the documentation from the software implies accuracy problems because of the evolving of both the documentation and software. But when should we create the class documentation is it before or after the implementation? Most of the class documentation is created after implementation for example the several automatic documentation tools using the class headerfile as input. We don't think the documentation should be created after the implementation. To our experience it is possible to create first the documentation of classes before the implementation i.e. step 1 document class X, step 2 implement class X. At the maintenance part we should be aware to keep the consistency between documentation and the code.

What should be exactly in the documentation of a class? The name of the class is the first attribute in the class documentation. The name of the class identifies the class but this isn't enough if we looked at the lifo-Queue-Stack problem described in chapter 4.1 To solve this problem we need a synonym attribute. Every programmer should be allowed to supplement this attribute (of course by consensus).
A class can be 'concrete' or 'abstract', if the class is an abstract class then the keyword abstract will be placed before the classname. The description of the class in the natural language should also be in the documentation of the class. The description of the class will give information about the behaviour of the class, but mostly it doesn't give the total working area of the object(domain), Therefore we introduce another attribute called domain. In the domain there is a sub-attribute called example. In this sub-attribute an example is given about the usage of the class.
The location of the class in the systems hierarchy is also important therefore we introduce an attribute hierarchy, where the hierarchy of the class will be given.
The object described by the class exists in a specific space. Sometimes we want to put constraints on the set of instances of a specific class in that space for example in a system we have the constraint that the number of instances of class Y is equal or lower than the value X. To document such constraints we introduce the attribute Set Constraints.
Creating an instance of a class it is possible that we have to fulfil some conditions, these conditions are valid during the lifetime of the instance. Documenting these constraints we uses the attribute Object Con-

straints. These Object Constraints can also be act as post-conditions at every method available in the object. The usage of pre and post-conditions are good tools to set up a class specification document. Using the pre and post-conditions at the implementation will also give provable code. In our document we will use the pre and post-conditions at every method. To specify the pre and post-conditions we use a mathematical notation. To describe the instance variables in the class we will use an attribute Instance variables. After every instance variable we will give a short description about the variable. If the class will be put in a specific library we have to find this information into the documentation. The author and the date is also important. Implementing all the information given in this paragraph results in the following document structure:

<*abstract*> **CLASS** Classname -- Description of the class.

SYNONYM -- Synonyms of the classname.

LOCATION -- Hierarchy tree.

INSTANCE VARIABLE
I_variable: I_variableType
description -- Description of the instance variable

METHODS
MethodName(Argument_1:ArgType, ..., Argument_n:ArgType):ReturnType
description -- Description of MethodName.
require -- pre-conditions
ensure -- post-conditions

OBJECT CONSTRAINTS -- Description of object constraints.

SET CONSTRAINTS -- Description of the set contraints.

DOMAIN -- Domain Description
Example -- The usage of the class.

LIBRARY -- name of the library

AUTHOR -- author and date of creation.

Due to the lack of time for this paper it was not possible to use this structure and thus to evaluate the class documentation. At the moment we define this as the first step for the improvement of class documentation.

6 Conclusion

Reuse of software is considered as an important quality ciriteria. Managers are still waiting for the time the software can be composed from prefabricated components. In this paper we have focused on the documentation problem of libraries. We have showed that the several class libraries were documented insufficiently. Our own approach try to reduce the insufficiency of the documentation by taking the good parts of the library documentation. We didn't focused on the use of tools because we believe a tool is just a tool and doesn't solve the document problem. The use of our class structure for documentation will show how applicable the structure is and if it is intuitive. Any comments/ recommendations on the structure can be send to the author of this paper. In the future we try to find a solution how the method problem can be solved in our structure i.e. should you only describe the methods defined in the class or should you also describe some method defined at parent or grand-parent level.

References

[Dumpleton], Graham Dumpleton, ‘OSE - A development environment for C++’

[Garlen] Keith E. Garlen, Sanford M. Orlow, Perry S. Plexico ‘Data abstraction and object oriented programming in C++’, John Wiley & Sons, 1990.

[Laverman] Bert Laverman, ‘Supporting Software Reusability with Polymorphic Types’, PhD dissertation 1995, Groningen University.

[Marciniak] Marciniak, John J. (Editor in chief) ‘Encyclopedia of Software Engineering’, page 1055-1057, Two Volume Sets, Wiley-Interscience 1994, ISBN 0-471-54001-8

[Meyer] Betrand Meyer ‘Reusable Software, the basic object oriented component libraries’, Prentice Hall,1994.

[Naher] Stefan Naher, Christian Uhrig. ‘The LEDA user manual’, Version R 3.2, supported by the ESPRIT II Basic Research Actions Program.

[Pace] Marco Pace, ‘Evaluation and comparison of C++ Class Libraries’ 1992

Decoupled Message Handling and Broadcasting Messages

Thomas Lidström
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: pt91tl@pt.hk-r.se
www:

Abstract

Often an object wants to send a message (call a method) to an object that possesses a specific role. The receiving object itself isn't of such interest as the role it possesses. But it is hard to specify that it is the role the sending object is interested in. And to make things even worse, what will happen if the object that possesses the role is replaced? Broadcasting messages, that is one object sending a message to many receivers, is a feature that is hardly used because current object oriented languages are based on message passing concerning only two objects. This makes broadcasting features hard to implement, at least in an intuitive way. In this paper these problems are discussed and different solutions, to the problems are suggested. This paper is organized as follows. Section 1 is a discussion around the role - object problem. Then three different solutions are presented in section 2, where the last one is the most interesting one. Broadcasting messages to many objects are treated in section 3. A conclusion is the given in section 4. Throughout this paper, OMT's[1] notation will be used in the figures.

1 Introduction

Message handling in current object oriented languages are mainly focused on that it is two objects involved in the communication, the object that is sending the message and the one that is receiving it. The sending object must in some way know where to send its message, i.e. it must have some knowledge of the receiving object. This is often realized in object-oriented languages by letting the sender object in some way have a reference to the receiving object, e.g. a pointer. This reference creates a rather strong coupling between the sender and the receiver. Through this reference the sender looks at the receiver as both an object and as the possessor of a role. But often the sender of the message isn't interested in the receiving object itself, but more the role the receiving object currently is possessing.

For instance, an object that wishes to send data to an object on a remote machine, perhaps needs to ask for permission to send. This could be done by asking the communication administrator, possessed by aNetworkManager1 in this case. This means that the object that wishes to send the data is more interested in the role "communication administrator" than the object that possesses this role.

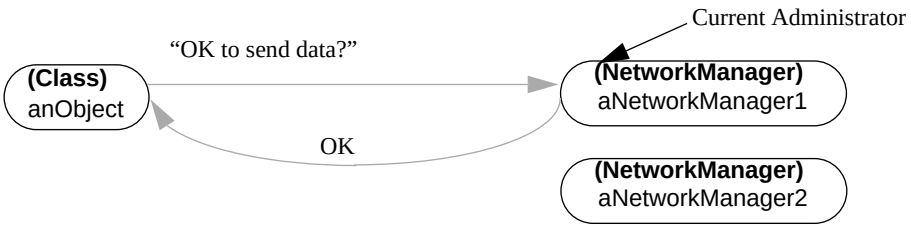


Figure 1. An object asking the possessor of the role for a service.

So far everything is fine, but what will happen if the current administrator is replaced by another object? Suppose that aNetworkManager1 is replaced by aNetworkManager2 for some reason, e.g. another part of the network must be used. The sending object is still referring to aNetworkManager1 as the possessor of the communication administrator role and thus the message is sent to it. But aNetworkManager1 isn't the current communication administrator and therefore doesn't know what to do with the message.

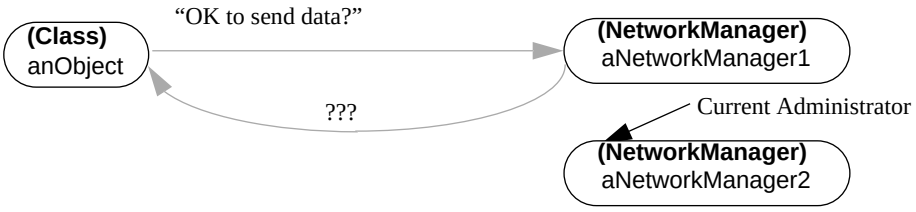


Figure 2. An object asking wrong object.

The problem is that the reference, referring to a role as well as an object, is hard to move from one object to another when the role is moved to a new object. Current object-oriented languages doesn't separate the role an object possesses from the object itself. Somehow either the object acquiring a role's services must be informed to move its reference, or the possessor of the role must be able to pass the reference to the object that is replacing it.

Another thing that doesn't make life easier is what to do if the current possessor of a role is replaced by an object that is of another (class) type. What will happen, in the example, if the network manager is replaced by a modem manager, because the network crashed? Sometimes this problem can be solved by having all possible role objects classes structured in a common inheritance hierarchy. But sometimes this isn't feasible or incorrect and the limitation still exists, how do an objects reach a role object that is of another type than the "original" possessor of the role?

And to make things more even more difficult, what if all the objects that acquires services from a role themselves are of different (class) types? This will complicate the moving of a reference at the role possessors side considerably. Somehow it looks like a role possessing class must have knowledge of all the (class) types that will communicate with it.

Except for these problems there are a few other practical problems that needs to be solved, namely how to pass arguments and how to receive the reply. Probably, the different role classes will perform different tasks, e.g. a network manager works differently than a modem manager. Thus they may need different information from the calling object, as well as their answers may differ. This will result in two minor problems, how does the sending object know what kind of information the receiver needs and how does the receiving object interpret the received data? The problem of how to receive the result of a message arises when the sender is unaware of the type of the role object. There can be differences in the reply received from a network manager object and a modem manager object.

These two problems will however not be of any greater magnitude as long as the coupling between a role and an object, possessing the role, is hard. It is when the coupling gets looser that the problems occurs. As long as the coupling between a role and an object is hard (e.g. they are the same) it is always easier for an object, acquiring its services, to know what arguments to send as well as what answers to expect. Moreover, these problems is also related on how strongly typed the language used is. The looser the language is typed, the greater are the risk of detecting the problem at a later stage.

2 Three Solutions

In this section three different solutions are given. Each one of them is described and evaluated.

2.1 Local Object Responsibility

One way of solving the problems listed above is to let the current possessor of a role having knowledge of every object that is interested in its services, as this role. That is, to have a list of references “back” to these objects. When a new object reports its interest to the objects services, a reference to it is stored in the list.

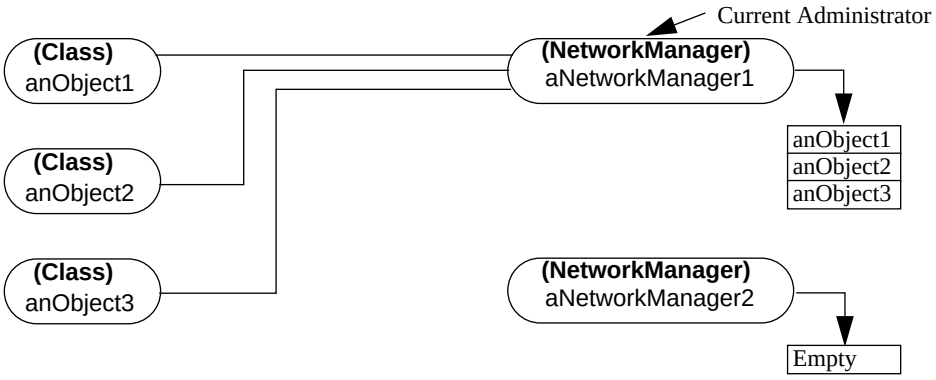


Figure 3. The local class responsibility.

In this way, the object possessing easily can be replaced by another object. When the object is to be replaced by another object, it notifies all objects in the reference list that it is going to be replaced. The objects notified is supplied with a reference to the new role possessor through the notification, and can therefore change their own reference. When the notification is done, the object in some way gives the list to the new possessor of the role. This could be done by either copy the list, give it away or by switching lists. The latter way is not to recommend because the new possessor could already be possessing other roles.

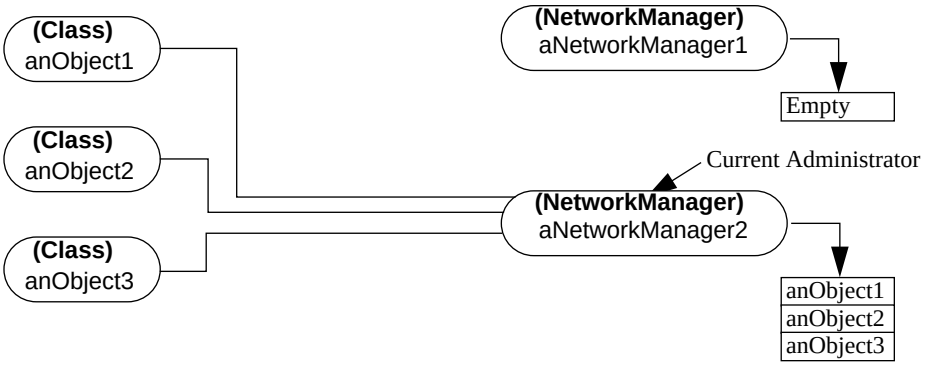


Figure 4. After a role object change.

There are a few drawbacks with this solution, one of them is the list of references. Maintaining such a list is both time consuming and requires more memory. Another is that if an object acquires services from more than one role, it will be in several reference lists. This is a somewhat redundant.

2.1.1 Different Types of Role Classes

If the classes that handles the role are of different types, and not in the same inheritance hierarchy, there will be problems when the role is moved from one type of class to another. If the object possessing a role is replaced by a new object of another type, the objects interested in the services will have difficulties because the reference to the role changes type. However, in strongly typed languages, a type error of this kind will

be detected at an rather early stage. And for loosely type languages there is a small possibility that no problems will occur, that is if the classes has the same method. Still, the only good solution is to keep all role classes in the same inheritance hierarchy. But this may force the designer to make trade-off's, and thus may not considered to be a good solution. This problem will probably constrain using a local object responsibility to systems that isn't of a great complexity, or at least doesn't consist of a large number of role classes.

2.1.2 Different Type of Callers

If the objects that acquires a role's services are of different (class) types, and not in the same inheritance hierarchy, there will be severe problems for the role objects, in maintaining the list of references to objects that are interested of its services. This because it is impossible to put different (class) types in a list, if they aren't in the same hierarchy. As earlier, the magnitude of the problem depends on whether the language used is strongly typed or not. In strongly typed languages it is not possible to put references to objects of different types in a list. In some loosely typed languages it is, and therefore it will be no problem at all as long as the classes has a similar interface. If they don't the role object will not be able to call them. Probably multiple lists will have to be maintained. Which, in turn, will increase both the complexity of maintaining the references as well as increase the memory used.

2.1.3 Passing Arguments

Due to the "hard" coupling in the relation between the concerned parts, the message sent will be made as an explicit method, and thus no problems with arguments should occur. The only time when there is a possibility to an error to occur is when the receiving object, possessing the role, is replaced by an object of a different (class) type. Then there may be a possibility that the arguments are to few or erroneous, depending of the functionality of the replacing object. E.g. if a network manager is replaced by a modem manager, a telephone number is most likely to be required instead of a host address.

2.1.4 Returned Data

The data returned will be awaited, the sender knows what will be returned and the receiver knows who called it. As before, the only possible risk of problems is when a role object is replaced by an object of another type, that has a different functionality.

2.1.5 Pros and Cons

By having the role objects themselves as responsible for the handling of the role is a quick and relatively simple solution. It will be sufficient if there are few types of both role classes and classes acquiring the role based services. But if the system gets to complex or contains a lot of either roles or objects acquiring the roles services, this solution will not be sufficient.

+ An simple and easy solution.

- Overhead implementation, maintaining the lists of references.

- Redundant information, the list itself.

- Hard to realize if the system is complex and/or there are many role classes of different types.

2.2 A Special Super Class

In the previous solution, every object possessing a role had to maintain a local list containing references to all objects interested in its services. This resulted in some redundancy. If the objects acquiring services, acquires services from many role objects, a reference to them will occur in every role objects list. Plus,

every role object had to maintain its own list. Another solution is to put all the role objects in one list, and have the list maintained by a super class to all the role classes. That is, to let every role class be a sub class to a common super class and in this super class have a list facility, maintaining the roles. In this way the list handling is centralized and put outside to role objects responsibilities. It is now a super class that keeps track of the current role possessing objects. In the picture below this is done by a enumerated variable.

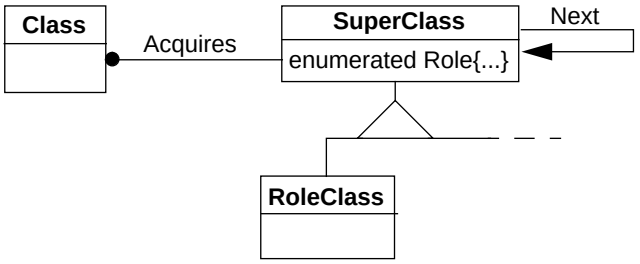


Figure 5. The super class solution.

All communication between objects acquiring role based services and the objects possessing these roles is done through the super class. The communication is done with class methods (in C++ a static method located in the super class). When a objects acquires services all it does is call the super class. Because all roles is accessed through the same class, the calling object in some way have to inform the super class which role, and service, it is interested in. Some kind of protocol would be proper. It is then up to the super class method to find the current possessor of the role and delegate the method call to it. It is now the enumerated variable comes in. It is through the enumerator the role possessor is identified. The reason for having a enumerated variable instead of a boolean is that with an enumerated, it is possible to have multiple role within the inheritance hierarchy

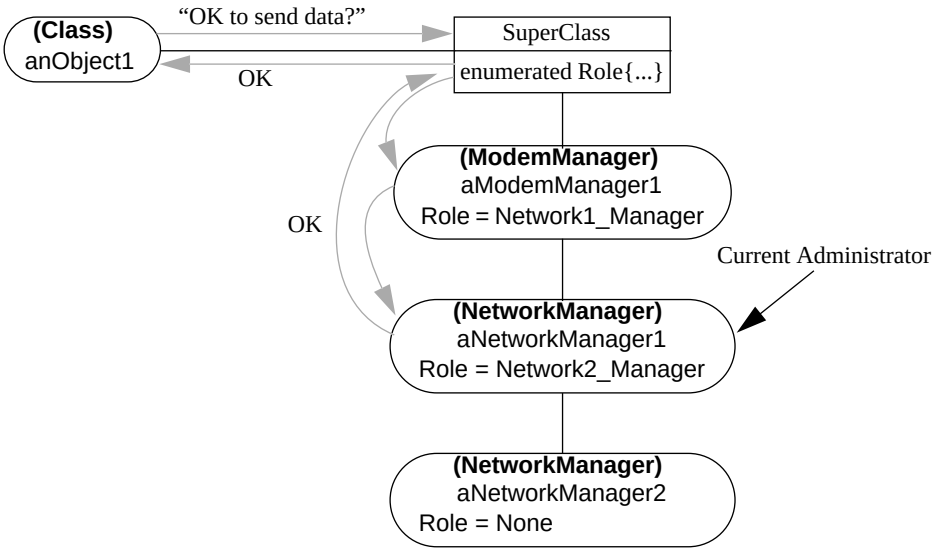


Figure 6. An object wants a roles services (The dotted line shows the execution path).

Every time a new role object is instantiated it is put in the list. If the new object is instantiated as a possessor of a role, this is stored in the enumerated variable. If the object is to replace another object, as possessor of a role, it does it through the super class.

2.2.1 Different Types of Role Possessors

Because all different (class) types of role classes will have a common super class, and the communication takes places through this super class there will be no problems when a role objects is replaced by another type of object. That is as long as all the role classes are within the same inheritance hierarchy, if not there may be severe problems.If the role classes aren't within the same inheritance hierarchy many inheritance trees must be used, making it impossible for an object acquiring role services to know where to send its message.

2.2.2 Different Types of Callers

By using a common super class, no "reversed" references will be needed and thus is this solution not affected by different (class) types of calling objects. Neither the role object nor the super class needs to know anything about the objects interested in the role's services. This is one of the advantages, compared to the local object solution, with having a common super class.

2.2.3 Passing Arguments

With this solution there will be no problems in passing arguments. There is still a relatively strong relation between the sender of a message and its receiver. But now the relation has moved from individual role possessing objects to their super class. Due to this, all sub classes will have to conform to the interface of super class. There is still a small risk for problems if the differences between the functionality of the different (class) types of role classes is large. But this is must unlikely to happen, if the differences are to big the classes shouldn't belong to the same inheritance hierarchy.

2.2.4 Returned Data

Because all the role classes are in the same inheritance hierarchy, the data returned from a method really shouldn't differ in such a magnitude that it should become a problem.

2.2.5 Pros and Cons

By using a super class to maintain the role objects some of the problems found in the local class responsibility will disappear. Instead of having every role object maintain lists, the super class maintains one list. But some problems still remains, and in some cases even have increased in magnitude. With this solution there will be severe problems in implementing several inheritance hierarchies using different super classes.

+ No references back to the callers are needed, compared to the local class responsibility.

+ A little bit more "intuitive" solution than the previous solution.

- The super class must have a variable in order to be able to differ between the different objects that possesses any role and those that don't.

- Still have a list to maintain.

- Very hard to have several super classes, offering similar role services.

2.3 A Separate Role Class

When using the super class there still remains some problems. Namely what to do when the role classes doesn't belong in the same inheritance hierarchy, and there still is a list to traverse every time an object wants a role based service. This partly has to do with the relatively strong coupling between the class

acquiring the role based services and the super class. The third solution suggested in this paper completely decouples the message handling between the two parties by putting a separate role class in between. Instead of having the objects interested in a role's services hooking a reference to an explicit object or a class performing the role, it has a reference to an explicit role, which in turn has a reference to the object performing the role based service. In this way the actual coupling between the two parties is the maintained through a the role. There might be a problem in how an object will know how to reach such a role object when the object is instantiated. Therefore adding a role handler class might improve the situation. The role handler has a list containing all current role available.

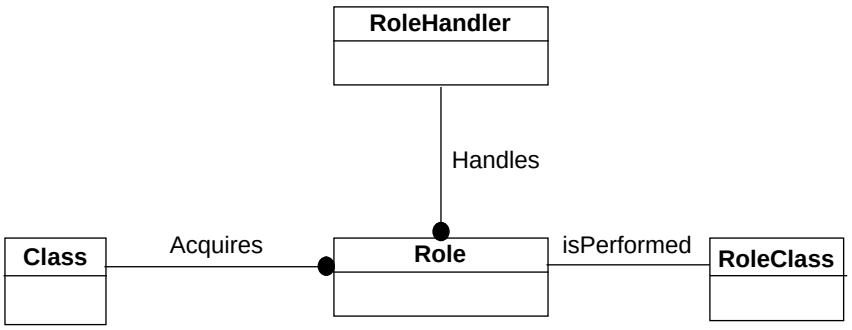


Figure 7. Object model showing the Role Class hierarchy.

When an object wants a specific role service it send a message to the role handler. The role handler receives the request and scans through its list of current roles. If an appropriate role is found, a reference to it is returned, else a failure is replied. There still is a list to maintain, but now the list is only used when object wants to have references to roles, not every time they use the roles services.

In a similar way, the objects providing the service offers its services to the role handler. Every time an object performing a role is created it contacts the role handler and the role handler puts it in the list of available role services.This could be done in two different ways, either the object itself creates a role object as well or the role handler creates the role object. If the object performing the service creates the role object there is a possibility that it creates a role object that already exists, there is already another object responsible for the role in question. But on the other hand, if the role handler doesn't know what role services the newly created object performs it will not know which role object to create. Which way to choose depends on how the role handling facility is implemented.

Now, when an object wants a service it simply sends a message to its reference to the role. All the role object does is to delegate the service call to its own reference to the object performing the service.

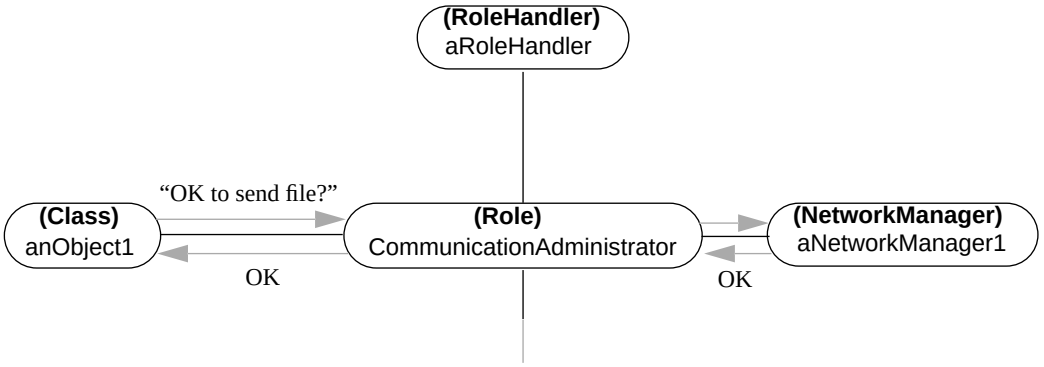


Figure 8. A service acquirement (The dotted line shows the execution path).

As can be seen, in this way the object that performs a role easily can be replaced by another object without bothering the objects acquiring the role services. In fact the replacing can be done in two ways, either by the role handler, or by the object itself. There shouldn't be any greater differences between the two ways, the replacing of an object performing a role shouldn't concern the objects acquiring its services. But one thing that might become a problem is what to do when an object performing a role is to be removed, without any object replacing it. The object acquiring the services will be unaware of this incident, due to the decoupling, and may very well try to send a message through the role object. The best way of solving this problem would be to place a check in the role object. If the role object has a reference to an object performing the role it delegates the message, if not it returns a failure.

2.3.1 Different Types of Role-possessors

If the classes that performs the roles are of different (class) types there will be no problems, as long as they still can be used by the role class. If the different classes should be in different inheritance hierarchies, then the role class will have to be sub classed into different types. One per inheritance hierarchy of the role classes.

2.3.2 Different Type of Callers

With this solution there will be no problems concerning different (class) types of role service acquirers., as long as they can use the role objects. As with the super class solution, the reference back to the objects acquiring the services are no longer needed.

2.3.3 Passing Arguments

There is a small possibility that there will be problems due to the fact that the object wanting a role based service don't know how to communicate with the object performing the role. It simply doesn't know how its interface looks like or how it would like to have its data. This could be solved by forcing all the role performing classes to have a common interface to their methods. That is, some kind of protocol must be used. Or a proxy object could be used to send the data, then it will be up to either the role object or the object performing the role to interpret the proxy. But there will still be a possible source of problems, if the functionality of the role classes differ in such a way that they need different data in order to perform their tasks.

2.3.4 Returned Data

How to deal with sending data back to the caller will have to be solved in a similar way as with sending arguments. The best way will probably be to make use of a future object. The object acquiring a role based service sends a future together with the other arguments and then the object performing the role puts the data to be returned in it.

2.3.5 Pros and Cons

By introducing a role- and a role handler class the coupling between an object acquiring a role's services and the object possessing the role is minimized. But instead a more complex message handling facility must be used, and in some cases it might be to much work to do.

+ Ability to completely decouple the message handling, i.e. the sender doesn't know anything about the real receiver.

+ Decouples the role from a object.

+ Ability to broadcasting a message, see chapter 3 below.

- Requires a somewhat complex class structure.

- Unable to add new kind of subscriptions run-time (in strongly typed languages)

3 Broadcasting Messages

3.1 General

Broadcasting messages is something that in many cases would have been very useful, if it was possible to implement it in current object oriented languages, in an intuitive way. Many times object orientation is compared with the "real" world, but when it comes to broadcasting messages it comes short. In the "real" world broadcasting messages is used every now and then. Communication between people is just one example. If message broadcasting could be easy to use in object orientation, many nice features could be done, in simple ways:

- Multiple use of the same information, the content of one object is sent to many objects, e.g. to show the same information in multiple windows.
- Synchronization, one object synchronizes many objects, e.g. before a global update of information.
- Security, if something illegal is happening, many objects gets to know it.
- The ability to subscribe to certain events within the system, e.g. every time an object of type x is created let all objects of type y and z know it.

There are however some problems that has to be dealt with. How to reach objects that are of different (class) types and how to deal with the answer returned from them. In most broadcasting implementations some sort of list containing references to all the objects that shall be called is used. If these references are of different (class) types, and not within the same inheritance hierarchy, multiple lists must be used, which will increase the complexity of how the broadcasting is done. Regarding how to deal with the answer, suppose one object sends a message to ten other objects and it expects every one of the to send a reply. How shall this be done?.

As mentioned earlier in most broadcasting implementations the object sending the message must send its message multiple times, to every object receiving the message. It would be nice if the object sending the message just had to send it once.

3.2 Broadcasting Messages Using a Handler Class

It is easy to see that the role handler facility, mentioned earlier, makes it possible for one object to broadcast a message to many receivers, if the semantics of the role handling is slightly changed. Instead of passing service requests from a sender to a receiver it has to be changed to pass messages from one sender to all objects interested in the message. This is easily done, by first changing the names of the concerned classes to more appropriate names and then by allowing many objects to be referred to by the same object managed by the message handler. If a message handler is used, that handles the messages that shall be broadcast, the object sending the message will only have to send its message once. All they do is send the message to a message object. In the message object some sort of list are maintained that contains references to all the objects interested in receiving the broadcasted message.

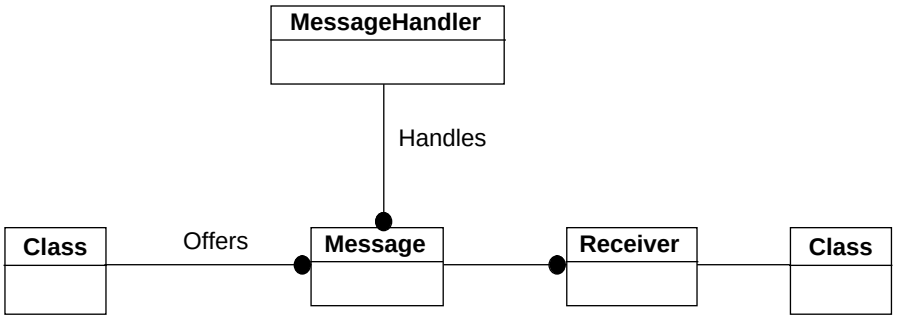


Figure 9. The Broadcasting class structure

Every object that has a message that it wants to broadcast notifies the message handler, which in turn puts it in the list of available messages. And all objects interested in receiving a broadcast notifies it by contacting the message handler. They are then put in the list of references maintained by that particular message object.

3.2.1 Different Type of Receivers

If the objects receiving a broadcasted message should be of different (class) types, and not are in the same inheritance hierarchy, the Receiver class will have to be sub classed. One sub class per inheritance hierarchy.

3.2.2 Hooking the Message to the Receivers

There is a slight risk to difficulties due to that the objects receiving the broadcasted message might not know how the message will look like. They must in some way conform to the message sent and have knowledge of what arguments to expect. This is probably best realized by using a proxy object.

3.2.3 The Answer Expected

The problem of how to receive the answer expected has also to be solved. Due to the fact that the objects receiving the broadcasted message will not have any knowledge of each other, they might overwrite each

others answer. One good solution to this problem is to have the sender to send a instantiated future along the message, and let the receiver out their answers in this future.

3.2.4 The Broadcasting

When an object has a message that it wants to broadcast, it sends its message to the message object. Along with the message it sends a future and a proxy, the future is for the answer and the proxy contains the arguments of the message. The message object then traverses its list of references and delegates the message to every object in the list. The receiving objects executes the message, using the proxy to get the arguments and putting their answers in the future.

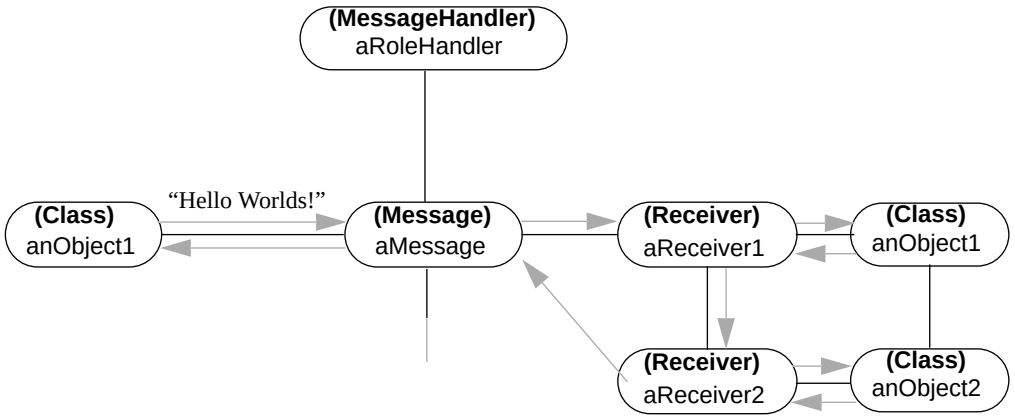


Figure 10. An object broadcasting a message to two objects (the dotted line show the execution path).

Meanwhile the sending object asks the future if it is ready, if not it waits. When the future is ready, and thereby all the receiving objects finished, the sender can start to examine the answer.

4 Conclusion

The usefulness of decoupled message handling and broadcasting messages is hard to use is current object oriented languages. These features isn't within the language, and instead they must be dealt with by the programmer, forcing him or her to do to solutions of various quality. By using handler classes the complexity of the decoupling, as well as the broadcasting facilities, are moved out from the concerned classes and into a separate class hierarchy. By doing this it introduces the ability to reuse existing solutions. Furthermore, handler classes isn't a new concept, but it hasn't got the attention it deserves.

Still the main problem remains, no object-oriented language supports neither role based message passing nor broadcasting a message.

References

[1] James Rumbaugh et al., "Object-Oriented Modelling and Design", 1991. Prentice-Hall, ISBN 0-13-629841-9.

[2] Doug Lea, "Roles Before Objects", 1995, WWW-document.

Identification in distributed systems

Magnus Östvall
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: mpt95mos@pt.hk-r.se
www: http://www.pt.hk-r.se/~mpt95mos

Abstract
We present a solution to the problem of identification in distributed systems. The solution addresses localization transparence and provides an environment which enables distributed applications to operate with automatic load balancing. In addition are subjects as distributed dynamic objects, method distribution and sharing as well as application sharing considered.

1 Introduction

In distributed systems there exist many new problems which does not exist in traditional computer systems. These problems must be addressed before stable distributed systems can be created. In addition to the distributed nature of the system there are also special properties which have to be achieved. This includes transparency of location and distributed system architecture.

This paper will first discuss distributed system examples with problems of identification and transparence. Our solution to these problems is presented and explained. The solution is then extended with features such as references to distributed objects and automatic garbage collecting techniques. Application sharing and concurrency is also discussed.

2 The identification problem

Identification in a distributed system has two main properties which has to be supported. The first one is that objects must be uniquely identified. The reason for this is that messages between objects shall be able to be sent across the distributed system. Another property is that transparency of geographic location in the identification must be met. The reasons for are that objects shall be able to be localized differently when load changes on the machines and that location dependent descriptions not shall be placed in applications.

2.1 Traditional distributed systems, two examples

Some distributed systems exist but they suffer from the drawback that some of the properties are not achieved. Two examples of large distributed systems will show this problem.

2.1.1 Telephone system

The telephone system is probably the largest existing distributed system. The solution have properties which are important in computer systems. The fact that telephone systems can either be analog or digital is of no importance, it's the identification solution which is considered here.
Telephone numbers in the local perspective is very simple. Each telephone subscription is assigned an unique number. The only characteristic is that each number must uniquely denote one telephone subscription. When the local area is extended into country wide areas the local areas are assigned a unique number

in the same manner. The total telephone number is combined of the local and the area code. The same procedure is applied in country codes.

The interesting property is the latest trend in mobile telephones. The numbering must now be constructed in a new way since there is no geographic placement of the numbers. The solution is simple, just use more digits so that all telephones can be uniquely referenced.

The telephone system numbering technique has two contradicting advantages. The first one is that no centralized numbering algorithm has to be done. Everything is taken care of in the subsystem. The second one is that transparency of geographic location is achieved. Unfortunately, the first advantage is only applied in traditional telephones and the second one only apply mobile telephones.

2.1.2 Computer identification

In traditional computer distributed systems, such as the internet, all machines are assigned a unique name (or number) and then the entire address is composed by larger and larger areas. For instance are often a special computer assigned a name of the form machine.department.university.country. This system suffers from the same problems as the traditional telephone system.

An practical example of this is the world wide web (WWW) pages which holds information, but all information is identified by the machine address combined with a file path. It would rather be desirable that the information would be accessible through the contents of the page.

2.2 Desirable improvements of traditional systems

2.2.1 Telephone system

The goal of all telephone systems is to enable someone to talk to some other (unique) person without knowing the location of the person. The goal is getting closer with today's mobile telephones, but there is still a number rather than the person itself that shall be called.

A improvement would be that a single (or multiple) telephone should not be referring to a unique person. The person should instead be able to be reachable by some method without providing localization information. A solution which is localization independent and still being able to provide telephone contact to a unique person would meet the properties.

2.2.2 Computer systems

WWW would be improved if the information would be localization independent in additional to access through the contents of the page.

2.3 Example analogies

The examples show that localization independence is very desirable in distributed systems. In the rest of the paper 'ordinary' objects is handled rather than persons or WWW pages. The same properties apply in any distributed system.

3 The aim of the solution

The main property we aim to solve is the problem with identification in a distributed system. A big issue is that transparency of location shall always be supported. The reason for this is that objects shall be able to be moved and that location dependant issues not shall be placed in applications.

One big property of distributed systems is that load balancing techniques shall be automatically used by the system. This is why objects shall be able to be moved (and perhaps be moved by themselves) to and from different machines.

The approach we have taken is that the distributed operating system already exist in some form. Instead have we set our focus on how to design distributed objects and how they will act together in a distributed system. The research covers ordinary objects and resources. In addition will method calls between distributed objects be discussed.

We have a concept called object hierarchy which the design solution is based on, the object hierarchy organizes objects in a distributed application. The hierarchy is not in the objects themselves, the corresponding object entry for a object is placed in the run time system of the application (created by the compiler).

All source code examples are done in C++ style with the additional change that assignment is done with ':='. The source code is not complicated and only shows small examples of applicable parts.

4 Identification

The objects are identified with a string which uniquely identifies the object in the code. The string could be converted into a sequence of numbers but for readability we will use plain strings. A object is identified with the nested scope and object name. The following example shows how objects in an application are uniquely identified.

```
class B       class C
{             {
   ...           ...
}             }

class A
{
   class B object1;
   class C object2;
   ...
}

/* Global */

class A object3;
```

The result of this is that object 1 is uniquely identified with the with the string '**object3.object1**' and object 2 is identified with '**object3.object2**'. The localization of the objects are not reflected in the identity and thus can the objects be placed wherever suitable in the system.

Methods are identified in the same way, as they are regarded as objects. If a method called method 1 were in the class C then the 'method object' would be identified with '**object3.object2.method1**'.

5 The location lookup

The object location of course have to be stored somewhere. The objective is to hide the location from the sender so that the receiver can be placed wherever it fit best within the distributed system. Each object within the system is associated with an entry of the following type:

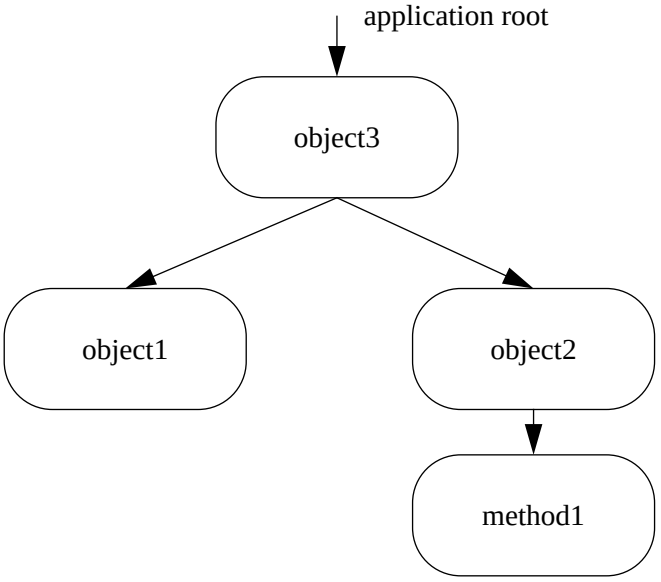
Table 1. Object entry

identification	mem	type	rights	parent id	parent loc	child id	child loc	...

Table 2. Object entry explanation

Entry field	Explanation
identification	Object identification.
mem	Memory adress of object on machine.
type	Different types of objects can be identified.
rights	Different objects can have different rights.
parent id	Parent object identification.
parent loc	Parent object location.
child id	Child object identification.
child loc	Child object location.
...	More childs can be stored.

In a possible implementation the object entries could be placed separately from the operating system kernel. This would enable different techniques for distributed applications to co-exist in one single distributed system.
A interesting property is that the object's location isn't placed in the entry.
The object entries now form the object hierarchy and we will show this by the following figure.



The object entries for all objects will be:

Table 3. Object entries

object3	mem	type	rights	root	root loc	object3.object1	object1 loc

Table 4.

object3.object1	mem	type	rights	object3	object3 loc

Table 5.

object3.object2	mem	type	rights	object3	object3 loc	object3.object2.method1	method1 loc

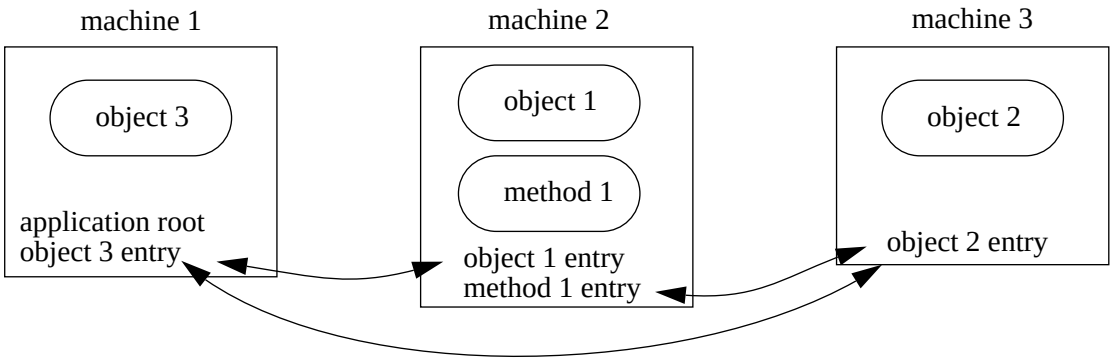
Table 6.

object3.object2.method1	mem	type	rights	object2	object2 loc

The application root holds the location for object 3 since there is no parent to it.

The location fields in the entries are depending on where the real objects are placed. The normal case could be that they all are on the same machine but this might not always be the case.

The hierarchy is represented by the object entries but the entries themselves are where the actual object is located. The following figure shows how the objects are localized when the objects are distributed.



The object entries are showing the direction of parents and childs.

6 Operations on the objects

The following section deals with creation, assignment and reference of objects.

6.1 Creation

When a object is created it can either be placed on the machine where the parent is located but a least load algorithm could be used to choose a better suited location. The distributed approach would enable execution on different machines but the communication will probably increase.

The steps necessary when creating a object are:

- Find machine to place object. Either same machine or remote machine.
- Create object entry on the machine.
- Update the parent object entry with the information about the child.

- Update the child object entry with parent information
- Create the object physically on the machine. (Object constructor is called.)
- Associate the name in the parent with the child identification.
- Return execution to parent.

6.2 Assignment

When assignment occur the steps are similar to creation but there can be differences.
The first steps are the same but when the object is created the assign method in the assigned object is called. After this the name is associated with the child identification and the execution is returned to the parent.

6.3 Reference

When an object is referenced then one approach can be to create a local copy of the object in the machine where the object which references the object is located. This approach can be very communication intensive if several copies of objects are created temporarily.
Consider the two following reference statements as an example.

```
class E
{
   operator+();
   ...
}
class D
{
   method2(class E);
   ...
}

class D object4;
class E object5;
class E object6;

object4.method2(object5);  /* references object 5, extra copying */

object6 := object5 + 1;    /* references object 5, copying ok */
```

The first reference statement would first copy the object 5, then call method 2 (which might be localized elsewhere) thus sending a copy of object 5 twice across the network. In this case, when the original object (object 4) never uses object 5, it's unnecessary to create a copy.

In the second case a temporary object is created on the machine where object 4 is located. (After the operator+ is called.)

In the first case a different way could be used. If a pointer or reference could be used as temporary object then the unnecessary copying of object 5 never occurs. This introduces new problems as we will show in the part about pointers.

7 Methods

Methods in general can complicate things in a distributed system. Objects, methods and resources are spread around in the distributed system. Method arguments might first need to be copied to the caller of a method, then some calculations might take place (which can be operator methods). The arguments then have to be copied to the machine where the method is located before the method actually can be called.

Return values are somewhat simpler but they might require copying operations before the value arrive to the caller of the method.

So far we have not considered whether methods are shared among objects of a class. This is the common way methods are used and might be desired if large numbers of objects often exist simultaneously. If each of these objects have an own set of methods the memory required will be great.

The solution would be that all methods of a class are shared among the all objects. This complicates things since the localization is not known by the object at creation time. The simple solution would be that the method localization is stored by the root object. This enables every object in the object hierarchy to create other objects.

If the method then shall be able to be moved from the localization it were when the objects were create the problem is caused by several ‘parents’ to one object.

There are two solutions for this. The first one is to keep track of every object of the class but this can be memory consuming since every method has to know about every object of the class. The other solution can be that the method informs the object root that the method is moved. There would then be a global update of all objects of that specific class. These solutions are desirable to improve since they either consume memory or time. This special issue is however leaved for future research.

8 Pointers in distributed systems

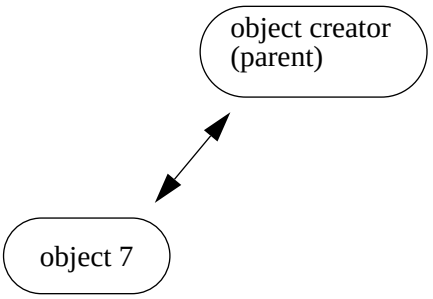
Pointers create one special problem in distributed systems. Pointers are changing and they can have multiple references, in the same meaning that methods can have several ‘parents’.

8.1 ‘Static’ references

Let us first consider the case when a object is created in the normal way, as an ordinary object.

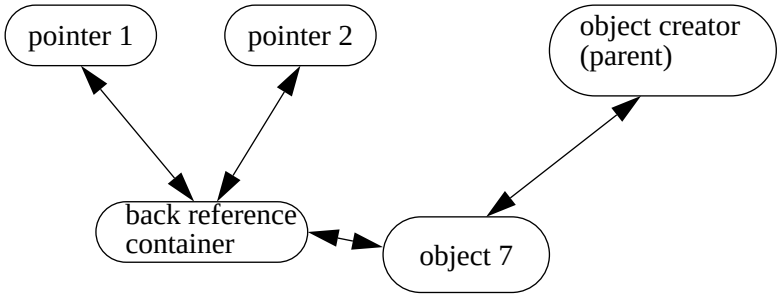
```
/* some creator object */
...
class F object7;
...
```

The object which creates the object 7 stores it as a child.



The object 7 is used by other objects and the best way to do this is to use references or pointers to it. This ensures that memory usage is minimized and that the object really is shared.

```
...
class F* pointer1 := &object7;          /* two pointers of type class F are created and */
class F* pointer2 := &object7;          /* assigned the adress of object7. */
...
```



The object entry for object 7 can be extended with a back reference container object entry. The purpose of the back reference container is to keep track of references and also for enabling automatic memory handling of objects which has no references.

In this example, when the object creator (the parent of object 7) deallocates object 7 the object 7 first is informed (or rather the destructor is called). The destructor could first check if there exist any references other than the parent. These references will be invalid if the object 7 is deallocated and they still have references to it. Instead shall the references be informed that their references no longer are valid. This could be as simple as setting the reference or pointer values to null or some predefined value.

This enables that no references exist that are referring an invalid object entry. (In traditional non-distributed systems this can be seen as no memory addresses exist which address unallocated memory.)

8.2 Dynamic pointers

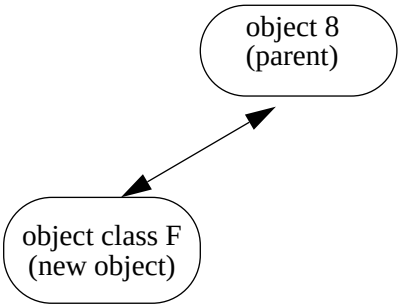
Pointers which are used with dynamic allocation and deallocation can also be used.

One solution with garbage collecting technique could be to have a reference counter [Collins 60] which counts references to the object. The garbage collection would work but the dynamic object will not be able to change localization without making the references invalid.

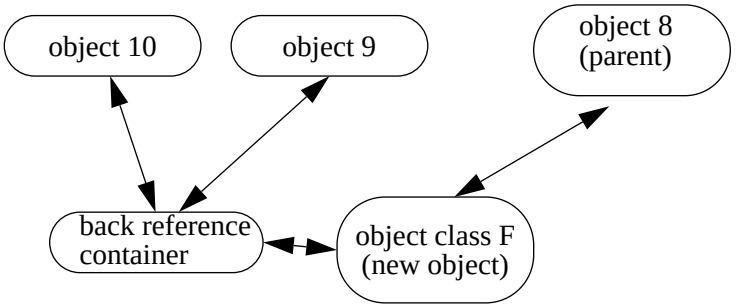
We will use the technique that the back reference container actually holds all references (back references). The following example will show that automatic garbage collecting can be achieved.

```
/* object 8 */
class F* pointer3;
...
pointer3 := new class F;        /* a new object is created dynamically */
...

/* object 9 */
class F* pointer4;
...
pointer4 := object8.pointer3;   /* another pointer references pointer 3 */
...

/* object 10 */
class F* pointer5;
...
pointer5 := object8.pointer3;
...
```

The object 8 first creates an object of class F (called ‘new object’ here), then assigns a reference (pointer) to the new object. The object 8 will act as parent to the new object, and will be references in the parent field of the new object’s entry.

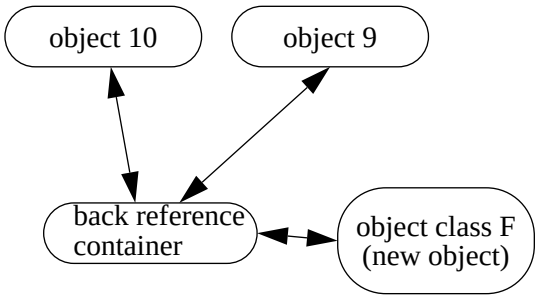


The next two events are that object 9 and object 10 also have references to the new object. The new object will have back references to the references.



8.2.1 Automatic garbage collection

The next thing that happens is that object 8 destroys it's pointer to the new object. (Note that the new object still exists.) The new object will be informed of the pointer is called. The new object will then have null values in the fields for parent and parent location.



The object 9 then destroys it's pointer to the new object in the same way. The reference back counter will now only have one reference to it. The next step could be that the object 10 now actually destroys the new object. This can be hazardous to do if the object doesn't know if there exist more references to the new object or not. (One solution to this problem could be that the new object ignored the delete statement if there existed more references to it.)

The other way the object 10 could behave is only to destroy the reference to the new object without explicit deletion of the new object. The back reference container will now discover that there are no references to

the new object and thus can call the destructor of the new object. This will ensure that no objects which have no references will exist.
This will result in automatic garbage collection, or rather that no garbage collection is needed.

All these reference constructions have drawbacks. For instance will many ‘back references’ exist but these are necessary for making the references (both non dynamic and dynamic) able to handle movements of the objects. Performance can also be suffering from this high number of references which has to be updated.
The ‘back references’ suffers from the same problem as reference counters, cyclic references will never be deallocated even if they are unreachable by the rest of the system [Mcbeth 63].
There are however big advantages of using these constructions, transparency of localization and automatic garbage collection of dynamic objects in a distributed system.

8.3 Identification of pointers

Identification of pointers is a little bit more complicated than ordinary objects. Ordinary objects have names together with scope as identifier. Pointers however can not use this technique. A simple method would be to use random numbers in the whole system. A pointer could then create an object with a random number as identification. This creates the possible problem with duplicated identities. The random number therefore have to be controlled to be unique. This could be done with a broadcast of the number and if it already existed on another machine, the machine would signal this back. This approach has two main drawbacks. A lot of communication and identification controls has to be executed whenever an object is created. Concurrency can also potentially cause two objects to have the same identification number.

The solution we propose is to use the scope where the dynamic object was created. This reduces the problem that other machines has to be involved when creating dynamic objects. The identification is however not complete yet. Since one object can create several dynamic objects in one scope some additional identification is needed.
The following example shows this.

```
/* some object */
...
class F* pointer;
while(some statement)
{
   pointer := new class F;
   ...                          /* the object created is not deleted */
}
```

The second object created can not have the same identification as the first object. The parent object which created the first object can not keep track of if the object still exist or not.

The solution is to add a sequence number to the identification. This makes every object identification unique. If the sequence numbers are made of, say 32 bits, a potential problem arises when or if they all are used and the value overflows to zero (0) once again. This is not regarded here since most applications don’t run that long or that so many objects are created in one single scope.

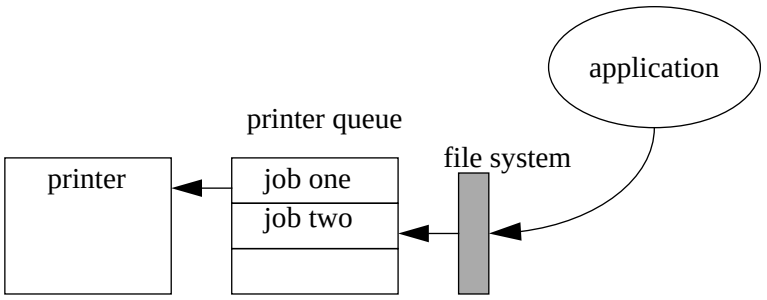
9 Resources

We choose to model resources in a distributed system as ordinary objects. Examples of resources can be database systems, printers or almost any kind of hardware.

The resources can be divided in two categories, those who are local for one machine and those who are accessible from the entire distributed system. The local ones are sometimes made accessible to the whole system. An example of this is the common printer which is connected to one machine in the system but all machines have access to it.

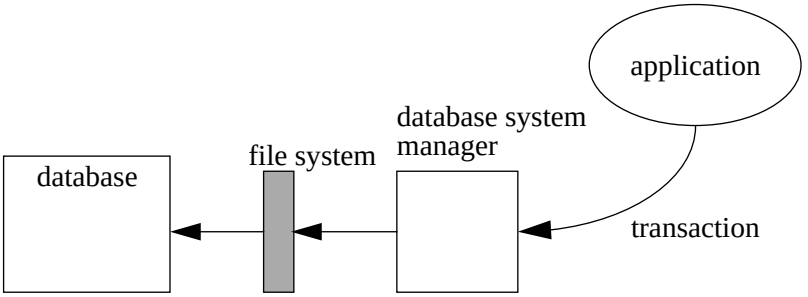
One approach is to have a daemon for enabling the printer to be used. The other way is to use the shared file system as a base for resources. Since a file system can be distributed in the whole system (such as NFS) [Tanenbaum 92] all machines can access it. If the resource is connected to some special type of file in the file system the resource can be used through the file system. (We assume the file system to support atomic actions.)

The following example shows how a printer could be used through the file system.



The queue is placed on the other side of the printer file so that all printer jobs are entered in the queue before being printed.

A database system does not work with this queue method since the transactions have to be atomic. The atomicity can be achieved if the database system is managed from the ‘front’ of the file system.

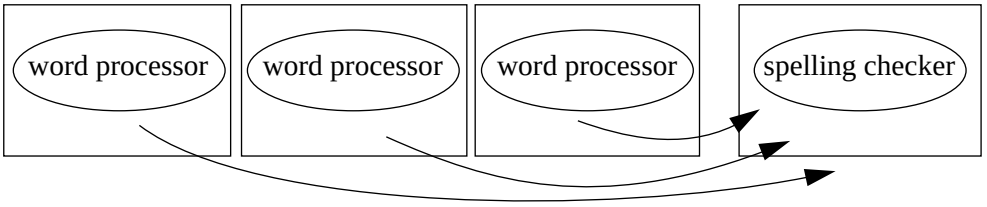


The database system manager selects and holds information about ongoing transactions. If the transaction is permitted it is send through the file system to the database. If the transaction have to wait because some other transaction has locked the database parts, then it’s up to the database system manager to handle this.

10 Application sharing

In a distributed system an application can have shared parts which all users could benefit from.

An example of this could be an ordinary word processor. The relatively small word processing edit kernel could be on multiple machines. When a new part of the processor shall be used, say a spelling checker, it would be better to send out a request to a shared spelling checker than to load the entire spelling checker on the local machine.



This technique must be combined with some kind of load balancing technique. The reason for this is that if suddenly every user of the word processor wants to correct their spelling it will increase the load of the spelling checker. It would then be better to load a spelling checker to the machines, or to create more shared ones which can handle the requests.

11 Concurrency

In distributed systems concurrency is a natural element which has to be considered. In our solution concurrency problems involve critical regions. Some parts of the system has to be atomic and protected by mutual exclusion methods.
Movement of objects and updates of object entries has to be mutually excluded to ensure valid operation of the system.
Every object is assigned it's own thread of control and messages send to and from objects has to be handled in queues or buffers.

12 Conclusion

The identification problem in distributed systems can be solved. Our solutions are not optimized for performance or memory usage. They are rather focused on achieving a certain feature in a distributed environment. The solutions also introduce new interesting problem domains. Future work would involve performance and memory optimizations, concurrency problems, method sharing in a class and also to test the solutions in a practical prototype.

13 Acknowledgements

We would like to thank Jan Bosch for research inspiration and Jessica Gillström for encouragement.

14 References

[Collins 60] George E Collins. 'A method for overlapping and erasure of lists.' Communications of the ACM, 2(12):655-657, December 1960.

[Mcbeth 63] J. Harold Mcbeth, 'On the reference counter method.', Communications of the ACM, 6(9):575, September 1963.

[Tanenbaum 92] Andrew S. Tanenbaum, 'Modern operating systems', Prentice-Hall International Editions, ISBN 0-13-595752-4.

BLoB

Andy Eskilsson
Department of Computer Science and Business Administration
University of Karlskrona/Ronneby
S-372 25, Ronneby, Sweden

E-mail: mpt95aes@pt.hk-r.se
WWW: http://www.pt.hk-r.se/~mpt95aes

January 12, 1996

Abstract

BLoB is the name of an portable distributed object-oriented environment supporting migration (phew, many buzz-words there) for programs written in Python, this paper describes some ideas for this environment.

1 Introduction

> `1 blob \ 'bl<a:>b\ n`
> 1a: a small drop or lump of something viscid or thick
> 1b: a daub or spot of color
> 2: something ill-defined or amorphous
> 2. blob vt blobbed; blob.bing :to mark with blobs SPLOTCH
>
> Websters dictionary

The name BLoB was chosen by taking the English word blob, and capitalizing some of the letters to make it look funny. Blob because its my view of an object oriented system, where all objects float around in the space of the system, a lump of jelly with objects floating around as seen in figure 1.

1.1 Why distributed and object-oriented?

One view of the real world is a number of nodes, that can host and work on/with objects, this is a normal distributed environment, a specialized node can host an object that can make use of, for example some special hardware on that node, and communicate freely with other objects around. Communication with the rest of the world is made through external connections.

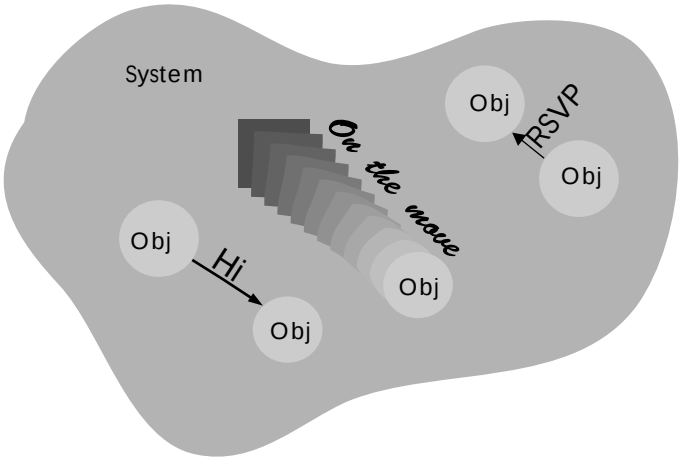


Figure 1: BLoB model view

Also in BLoB the migration of objects is included, objects can move around freely between the nodes (as long as the needed environment can move with them) (a node is a running python interpreter), therefore many nodes can reside on the same computer.

There are a couple of reasons for objects to be able to move around:

- Load balancing, it is easy to move objects from loaded machines to more unloaded
- The object needs some service that the current node can't provide
- The node that the object resides on might need to be alone for a while (for example taken down for hardware maintenance)

The reason for object orientation is that the encapsulation of data and methods makes it easy to move them around.

One way to use a system like BLoB, is in assembly-line like applications, where the object has to go through a couple of machines in a certain order to be processed.

Another usage could be some sort of file server, where the nodes could represent disks, when a disk get full, the storing object could move to the next node.

2 Distribution and migration

Distribution covers both the objects placement in different places and the migration of them, giving them the ability to move around, this leading to a couple of problems that will be take up here.

The main problem with distributing objects is how to know where they are, two solutions have been considered, either the reference to the object contains information of the original object, and when the object is moved it leaves a "I have moved" note.

The other solution is a server-based solution, all messages have to go through a server that redirects the messages to the correct object, this is the chosen solution because there already have to have a server managing the source code.

When migrating objects there are the possibility that the class defined by the object doesn't exist at the receiving host, therefore there exists a server handling source-code for class declarations. When a class definition is needed, the node asks the server who replies with the the source that is sent to the node.

Another problem that rises when handling a system with modifiable classes during runtime, is what happens when a class is redefined, when there are active objects of that type, in Python this is handled by letting the old object keep the old class definition, and all new objects to be created are cleared with the new definition, this is an approach used in BLoB too. When a class is modified, it is only the new instances of this class that are affected by the change.

There is one special case, when moving an 'old' object, the server needs to know what version to send to the receiving node, this can be accomplished by the server keeping a number representing the version together with the source of the declaration.

When the server notices that there are no more objects left with that version, it can delete the corresponding version.

3 BLoB

What does a running BLoB system look like?

Figure 2 shows a view of a BLoB system up and running, it consists of three nodes (python interpreters), five objects, one server and three controllers, all communicating through sockets.

Controller is the name for a external device attached to the BLoB system, acting as GUI to some objects (programs), for controlling and inserting new code into the system.

The server takes care of looking up messages sent between different objects, and a node is the execution environment for a node.

4 Choice of language

Python is chosen for the base platform of this system, mostly because Python is:

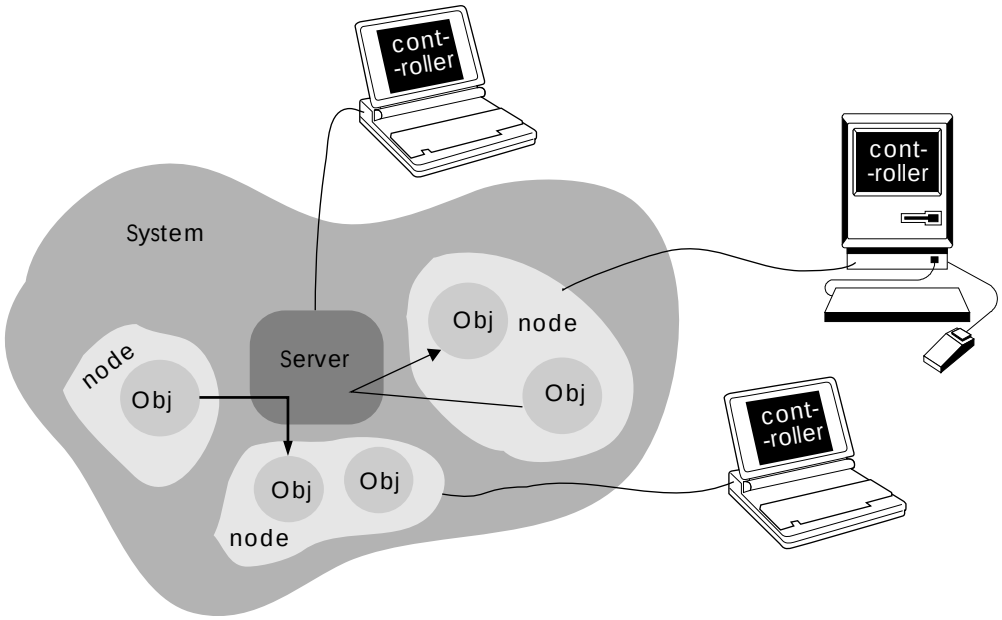


Figure 2: BLoB system view

Portable The Python interpreter is running on PC(dos/windows/win95/NT), mac and most UNIX platforms, almost all the platforms have socket support, and Python code is 100% portable under runtime, no compilations needed, unfortunetaly threading doesn't work on all platforms, but this doesn't stop BLoB from working in 'single-object-mode'.

There also exists a portable GUI that can be used for communicating with the system.

Extendability It is easy to, during runtime, add objects, object-types code, both by importing Python code, and through dynamic linking (on OS's that supports dynamic linkage) with object-code (for example c or c++) during runtime.

Interpreting It is possible to execute strings, this allowing us to bootstrap, sending object definitions, take a look at the state of the object by sending it a request in Python to interpret.

The interpretation in python also enables that the object doesn't need any special knowledge about for example what method it or some other object contains, until the call, this enabling an object to execute in an environment not supported by some methods, by not executing them, and if they are called the object can by itself verify the environment and migrate if needed.

Threadable Python supports threads on some platforms (for example solaris and AIX), this giving a good starting point for servers. Also this gives the possibility to keep several objects running on one node. Python contributes with a mechanism for mutexes so locking can be done between threads when needed.

Unfortunetaly the GUI-packages provided with Python doesn't work well together with threads, therefore the approach where the controllers lie outside the BLoB, and talk to the nodes/servers.

RTTI Python provides functionality for runtime type information, and storing/retrieving objects without any need for cooperation from the object itself, this easing up the migration a lot.

Exceptions Python supports exceptions, this easing up the lookup for methods to when it is needed, at execution time, if a method call fails, instead of aborting, the object can raise an exception and message unknown can be sent back.

Altogether this makes a system that can almost seamlessly run on the standard platforms provided at normal places, without any major modifications(reformatting hard drives and installing a new OS) needed as when using for example the Apertos operating system, you only have to set up a python interpreter for the platform, connect to the server and off we go.

5 Short introduction to the Python language

Python is a portable interpreted object-oriented language, with an elegant syntax and a small number of powerful high-level data types built in. The language can be extended by adding modules either written in python or other languages such as c and c++.

Programs in python can be generated quick, and it is good for rapid application. The infoseek search engine is based on Python.

With the python standard distribution there comes a large number of such modules for different tasks, such as interface to common unix databases, interfaces to GUI's, modules for making data persistent and much more.

Python is type less regarding parameter passing and it doesn't check for method/functions until just asked for it.

The main object-orientation issue of python is the inheritance the language provides, it supports both single and multiple inheritance from other classes.

One feature of Python that is very usable in the BLoB system, is the ability to check what objects/classes does there exist at the moment, and get much information about them.

5.1 Some code

Here follows some advanced code for a hello world program

5.1.1 A class definition

```
class HelloWorld:
   def __init__(self):
      self.Message=[]
      for i in range(1,5):
         self.Message.append("Hello World!")
      for i in range(1,5):
         self.Message.append(i)

   def printMessage(self):
      for i in self.Message:
         print i

   def errorMethod(self):
         self.unknownMethod()
         print "We shouldn't reach this point.."
         return "Ok"
```

What probably first catches glances is that instead of using some kind of begin/end symbols for defining blocks, python uses indentation.

This declaration defines a class called `HelloWorld` with the methods `__init__` (constructor), `printMessage` and `errorMethod`. All of the methods take only one parameter, `self` which is used to reference methods/data in the object (like c++'s this pointer). The `self` parameter is set automatically when the method is called.

The `__init__` method declares a data member of HelloWorld by first assigning it to an empty list [], and then inserting 5 strings containing "Hello World", and then inserting numbers 1 to 5.

Later on the `printMessage` prints all values in the list, and doesn't care what types there is in the list, just prints them.

The `errorMethod` is here to show that it is first at runtime this would generate an error, when we try to invoke errorMethod the program is halted at the call to `self.unknownMethod()`, it can be caught with an exception.

5.1.2 A Program

```
object=HelloWorld()        # the init method is called
object.printMessage()      # 5 hello world, and the numbers
                           # 1-5 is printed
object.errorMethod()       # This generates an error.
```

For more information about python, take a look at: `http://www.python.org`.

6 How is it done?

Python is used as a platform for BLoB. The programmer can choose between blending local objects (they could also be local to a global(distributed) object), handled as normal objects in Python, or having objects that are distributed.

One of the main advantages of having an external controller to check the system with is that the controller can be run with different interfaces, without BLoB noticing.

6.1 Towards Python

BLoB is an extension of python, not transparent, to use the functionality of BLoB the objects to distribute have to inherit from a base class and have to go through some special procedures when creating the object and sending messages to it. Python has a simple garbage collector that takes care of the objects not used any more.

Global objects are handled as references that are acquired upon creation, and can be used to communicate with the object and shared with other objects to spread the knowledge.

6.2 Towards the programmer

6.2.1 Calling remote objects (sending messages)

When sending a message to a distributed object in BLoB, a method in the reference object is called with the first argument as the method name and the following arguments as the arguments to be sent to the method.

All the arguments are handled as an array that is packed and sent to the receiving object, where the array gets unpacked and the first element is extracted as the method name, and the rest of the array is passed to the method (Python can handle argument list¡-¿array conversion).

6.2.2 Classes

Classes that shall be distributed have to inherit from a base class containing methods to handle message passing.

When declaring a new global class it has to be registered with the server, and also the declaration (source code) have to be sent over, so the server can provide it to parties that requires it, this can for example be done through a GUI if needed, or a node can send it by itself to the object.

6.2.3 The BLoB base class

The base class mentioned have methods for receiving messages, and forwarding them down to the class that inherits from it, it also contains the pack/unpack functionality.

6.2.4 Instantiation of a class

Instead of using the normal Python way (`instance=class()`) when instantiating an object a special method has to be called that returns the reference object, and registers the freshly created object with the server.

6.2.5 Programs

In programs the programmer have to use special routines to create global objects, and a special method when calling.

When a method is to be called in a global object, a request to the server is sent, where it either is redirected to the right object or if no such object exists, a message is sent back telling unknown object.

If the object is known, the server sends the message to the node containing the object, where the message will be translated into a string (in the BLoB base class) that is executed, trying to call the named method in the object, if it fails an exception is raised and an error is sent back to the server, who forwards it to the calling method.

It is also possible to ask the server to send a message to any object of a given class.

6.3 Under the hood

Each node will run a controller program, that listens on a port for commands from a server/external client, the server takes care of redirecting messages to objects, and controlling the movement of them.

7 The server node

The server-nodes main task is to redirect messages between the different objects in the system, and control the objects migration.

The server needs to contain data about:

- Nodes their ID, and what objects reside on each node, and the status of these objects.
- Class declarations, and version of class.

The migration is initiated by a request sent to the server telling what object to move where.

First the server checks if the destination supports the object, sending over the class definitions that might be missing. Then the server checks if the object is free, if not, all messages to the object is queued, and the server waits for the object to be free so it can move it.

When the object is free to move, the server asks the node to pack it and send it over as a string, the server passes it on to the receiving object who unpacks this string. The whole process can be 'guarded' by exceptions, if anything goes wrong, depending on what happened the migration can be cancelled or request for more data can be sent to the server.

8 Conclusion

It haven't been an easy task sorting out how to 'implement' the system, but the flexibility (absence of borders?) in Python have helped very much, you think you are stuck, and there appears for example functionality to handle arguments to a method as a normal array.

This paper haven't ended up in a running system, but all of the ideas in this paper are implementable i Python.

8.1 Layers

It is possible to implement something like the layers in LAYOM by hooking onto the message receiving method in the base class.

8.2 Security

Security isn't mentioned in this paper, but it could be easy to implement some kind of magic-cookie or public key check through the base class that handles all the message passing.

9 References

There haven't really been any identifiable references, the work is based on readings from Andrew S. Tanenbaum's Distributed Operating Systems, and the course material for Advanced Object Oriented Concepts.

Laguage Support for Small Applications to become a Big Cooperating System

Magnus Broberg
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: Magnus.Broberg@ide.hk-r.se
www: http://www.pt.hk-r.se/~magnusb/

Abstract
Today, home electronic devices are getting more and more advanced functionality and more and more computeral capacity. This leads too more expensive devices. In order to decrease the cost, both for the customer as well as for the developer these devices should communicate with each other and be using each others knowledge of the world as well as using other devices to perform actions.

This is a new area where small computers are helping each other, and rises new questions to be solved.

1 Introduction

Today there are a lot of problems with distributed systems, it seems to focused on a number of computers running one or more programs independently of each others, as in [Tanenbaum 92]. It seems likes the idea of having a distributed system there different parts of the system asks other parts for help solving a task. Distribution is weak in traditional object-oriented languages [Wegner 90].

In order to come a little closer to a truly distributed system this paper was written. This truly distributed system is for example a system consisting of a number of ECHELON nodes, each with its own application communicating with each other in order to solve a problem. One important this is that one application on a single node can not solve the problem by itself it have to require help from another node. This is the distinction of our definition of distributed system and the commonly used, for example in [Tanenbaum 92].

Since this system is manufactured by several manufactures there must be a way of restricting the use of services that a node offers. This was noticed as early as in the 90's that the need for different interfaces was needed for different callers [Wegner 90]. But there was no solutions.

In order to keep as good service as possible from the users point of view, there must be a way of working on solving the problem while waiting for another node to send an answer back. Or even start executing a new task initiated by the user while finishing the old one. This is done by Delayed answer, which is nothing new, [Wegner 90] and [Micallef 88], but no solutions was found.

2 Outline of this paper

This paper is based on the C++ language, with some extensions in both syntax as well as semantics.

The remainder of this paper describes Our domain as the first chapter, which defines what problem can occur. The chapter Macro objects which deals with a definition of the different parts in the distributed systems. The next chapter One Process / (macro) object specifies a scheduler of the objects and the scheduling of executing methods. Chapter Real Messages defines what is needed to send over the network in order to execute a method in a remote object. The chapter Delayed answer concerns about the potential paralleliza-

tion which is possible with having several applications running on separate hardware using the scheduler in chapter One Process / (macro) object. The chapter Security defines the different strategies for preventing unauthorized use of an object using the information given in chapter Real Messages. Finally Future work and drawbacks consider what is not yet done but would be nice to have...

3 Our domain

Humans need for small electronic devices which help her in the home seems never to decrease. In order to decrease the cost for each and every single device to contain all possible features, the device could use the knowledge from other devices. This will reduce the cost (both list price as well as development). In that order all these devices are objects in a large distributed system. Devices from different developers must also be able to communicate with each other. The communication network will probably be the power supply network or a network in parallel with the power supply network.

Below are some aspects that are important in this domain (which could also be stated as goals):

- Definitions of which types of devices that exists.
- What kind of services the devices are performing, and how to use them.
- The internal implementation of the device should be unimportant when using a service.
- Passing requests from one device to another.
- Ability to perform services to the user as quickly as possible, even multiple services while performing another in order to give quick response.
- A controlled way of assuring that devices are not supplied with too detailed information or even some kind of classified information.
- Prohibiting intruders to get classified information.
- Allowing only the developers to get services from their own devices, prohibiting other developers of similar devices using some services.

4 Macro objects

One problem in this domain is that the application is actually all the small devices scattered all over the area together, i.e. light bulbs, switches, dishwashers and the electrical hair dryer. In another point of view each of these devices is an application communicating with each other. In order to define what an application is the concept of macro object is introduced.

It is important to define the difference between a macro object and a “normal” object. A macro object is a single device, i.e. a certain light bulb, belonging to the macro-class light bulb, which in turn is a subclass of the macro-super-class light. The macro-super-class light could have another marco-sub-class called strip light. The macro object is mapped to a single traditional program running on it’s own with own CPU, memory and hardware, using ECHELON’s notation a macro object is mapped to a node consisting of a neuron chip. The program in our domain is the total number of macro objects running together.

This approach gives the ability to define an on/off switch for the macro-super-class light used by switches on the wall. Meanwhile the macro-class HiFi also have an on/off switch, it doesn’t belong to the light class and can be distinguished by the switches on the wall because of the different type.

The macro-class is a specification of different types of application and does not have any code. This specification also incorporates what objects must be inside the macro object in order to belong to a certain macro-class. Inheritance (on macro-class level) is simply a to redefine and/or extend a previous specifica-

tion. The encapsulation of objects is done in this specification, simply by defining which objects (and their methods) should be inside the macro, how their interface will look like and what they will perform. How these “normal” objects are implemented is the developers problem, just as using hidden objects to be used for internal use only.

A macro object could be considered as specialized client categories [Bosch 95], with the restriction to be pure in the sense of only consisting of layers. Further on a macro object is as mentioned above a single running program with capability to execute methods in other macro objects. The macro object is in other words a logical collection of several objects building a program to reside on a single hardware.

In the case of using ECHELON’s technique using a neuron chip, a macro object is a program running on one single neuron chip. Thereby each macro object is individually addressable over the whole planet. The name of the macro object is effectively the neuron chip address.

A macro object contains a number of “normal” objects inside it self. See *Figure 1. "Objects inside macro objects"*.

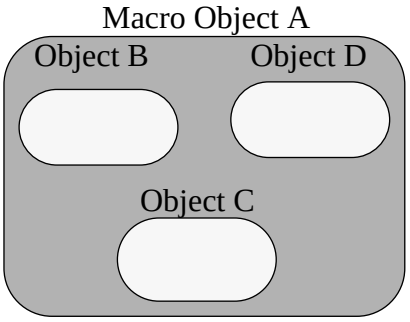


Figure 1. Objects inside macro objects

These macro-class specifications could be defined by some world wide standardisation committee, probably defining two or three incompatible standards to choose from.

5 One Process / (macro) object

The reason why not on every start of a new method as an own thread is of course the existence of race-conditions. Consider the following case:

```
class Account
{
   public:
      InsertMoney(int value);
   private:
      int moneyInAccount;
};
Account::InsertMoney(int value)
{
   moneyInAccount = moneyInAccount + value;
}
```

If we call Account::InsertMoney twice with the two threads of control the first thread may not have assigned the new value of moneyInAccount before the next thread is using the variable for reading, further reading about race-condition is found in [Tanenbaum 92].

The next step is to construct lock so that a method automatically locks all instance variables, this could be done in a non-dead-lock way by having a lock-hierarchy always locking in the variables in a specific order,

i.e. the alphabetic order of the names of the variables. This approach seems very attractive, but consider the case of recursion (it may not be a direct recursion) see *Figure 2. "In-direct recursion."*.

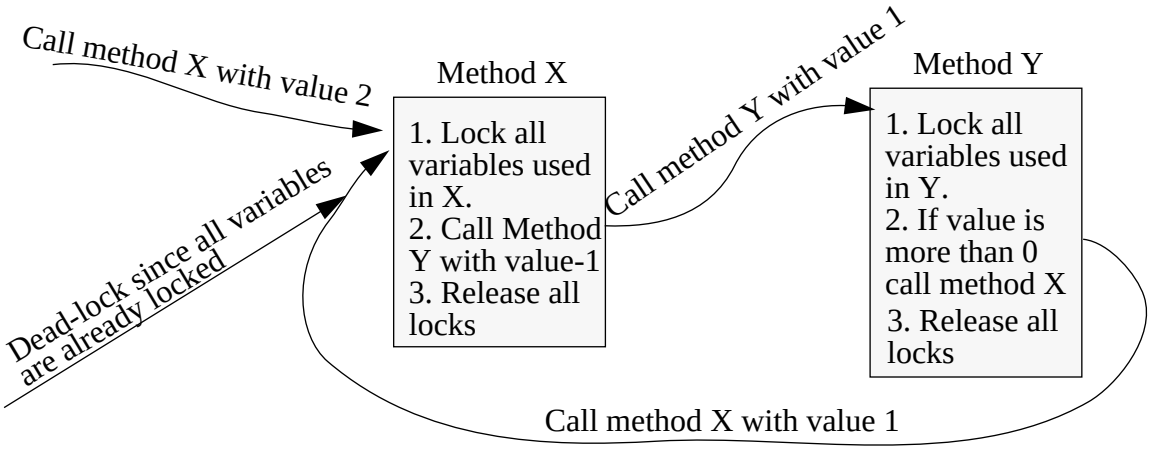

Call method X with value 2
Method X
1. Lock all variables used in X.
2. Call Method Y with value-1
3. Release all locks
Call method Y with value 1
Method Y
1. Lock all variables used in Y.
2. If value is more than 0 call method X
3. Release all locks
Dead-lock since all variables are already locked
Call method X with value 1

Figure 2. In-direct recursion.

This kind of recursion is hard to deal with since you can't tell whether the method you are using will call the calling method again. Methods X and Y can be in different objects or even different macro objects.

Yet another problem with locking is that a class may have references (pointers) to other object (in even another macro object), should the whole object be locked which will reduce the concurrency over a longer period than necessary. Also the locked object needs to lock its objects and so on.

As a conclusion, locks are not useful in this particular domain, but on the other hand a totally lock-free situation is also not applicable. How to deal with it?

One solution is to let the programmer explicitly state that we have a risk of having race-conditions as in [Bosch 95], but since the domain is to have a macro object bought off the shelf running for several years without any problems we cannot accept that approach.

The solution is (unfortunately) to decrease the concurrency within an object to contain only one thread of execution per object. If the macro object is running on a small system (slow CPU and/or little memory, because it is cheaper) then the whole macro object may have only one thread of execution, because it may not be a multitasking kernel developed for that kind of platform.

When an object receives a message it evaluates the message-identifier and all parameters. If a method is already running in the object, the start of the new method is done as soon as the old method is finished or blocked (see *Delayed answer* on page 8 for further explanation, at this point a method is being blocked whenever calling another method or itself). Whenever a previously blocked method is runnable again it will continue its execution as soon as the thread of control is available.

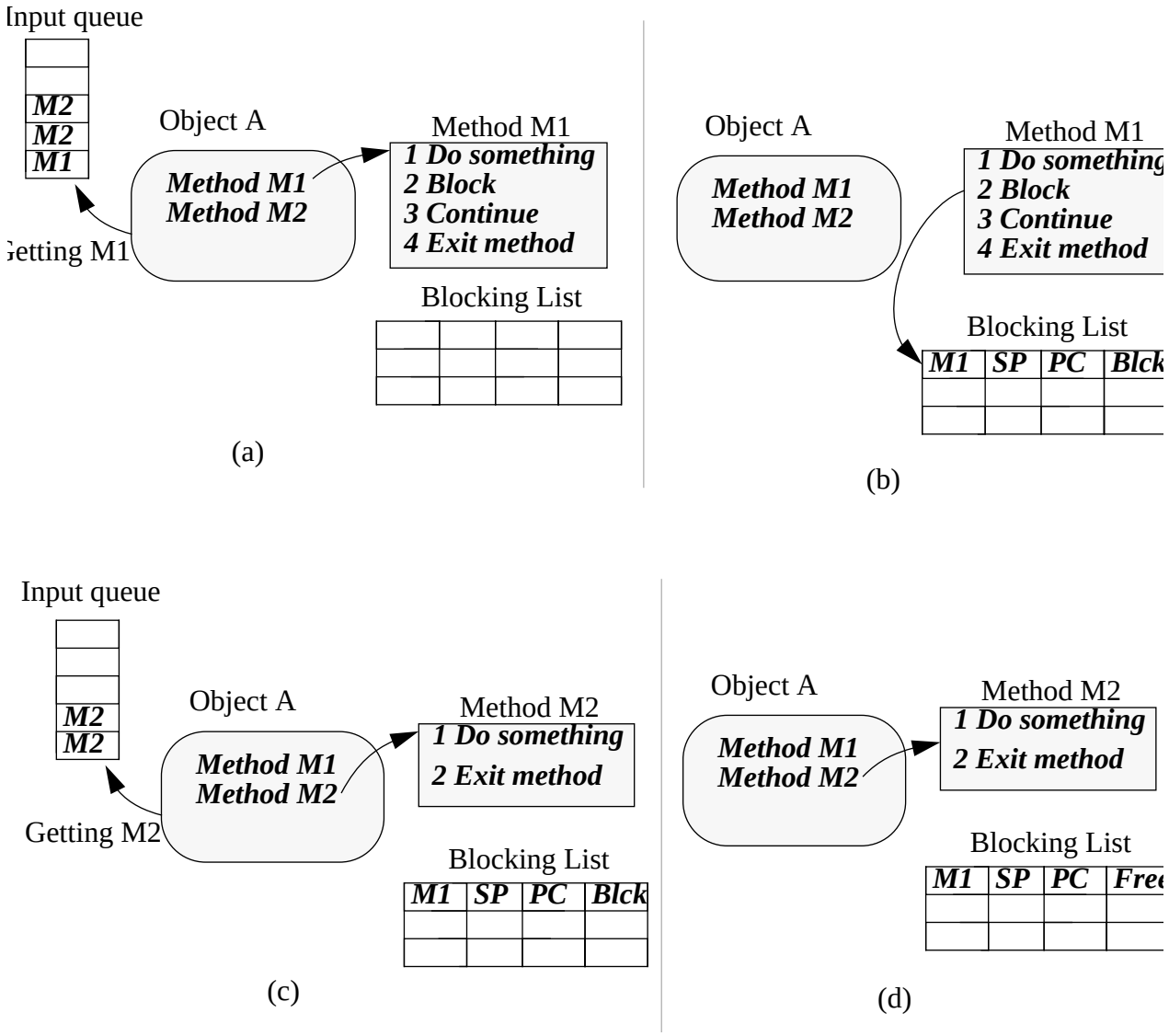
This requires an input queue per object for incoming messages. This approach is partly found in [Smolka 92], where messages are received one by one, thus guaranteeing a mutual exclusion property. How this in practice is managed is not mentioned.

As soon as a message being successfully executed the next message is being processed by the object. In the case of an method being blocked (see *Delayed answer* on page 8 for further explanation, at this point a method is being blocked whenever calling another method or itself) when a message is being successfully processed the list of blocked method is searched to find a blocked method that have been unblocked and is ready to run during the processing of the previous method. This gives a two level priority, the least priority has the new methods calls and highest priority on the messages that has once being started but was blocked for some reason.

Figure 3. "Scheduling within an object" will show all this in detail. In (a) *object A* selects message *M1* to process since there are no waiting methods in the *Blocking List*. When processing method M1 it will be blocked, and in (b) put it self in the *Blocking List* along with the methods *Stack Pointer* (SP for short) and its *Program Counter* (PC), finally a flag indicating that the method in *Blocking List* is really blocked.

In order to continue the processing of *object A* it select to process method *M2* which is the next method call in the *Input Queue* since there are no runnable methods to process in the *Blocking List*, this is illustrated in (c). During the time of processing *M2* the method *M1* becomes unblocked (or *Free*) as in (d). This is marked in the *Blocking List*.

Finally when *M2* has finished processing, the *object A* will try to find something new to process it will begin by looking in the *Blocking List* and finds that *M1* has been set free and starts executing it (e). At last when *M1* has finished processing object A selects to process the second method *M2* because there are no methods waiting in the *Blocking List.*



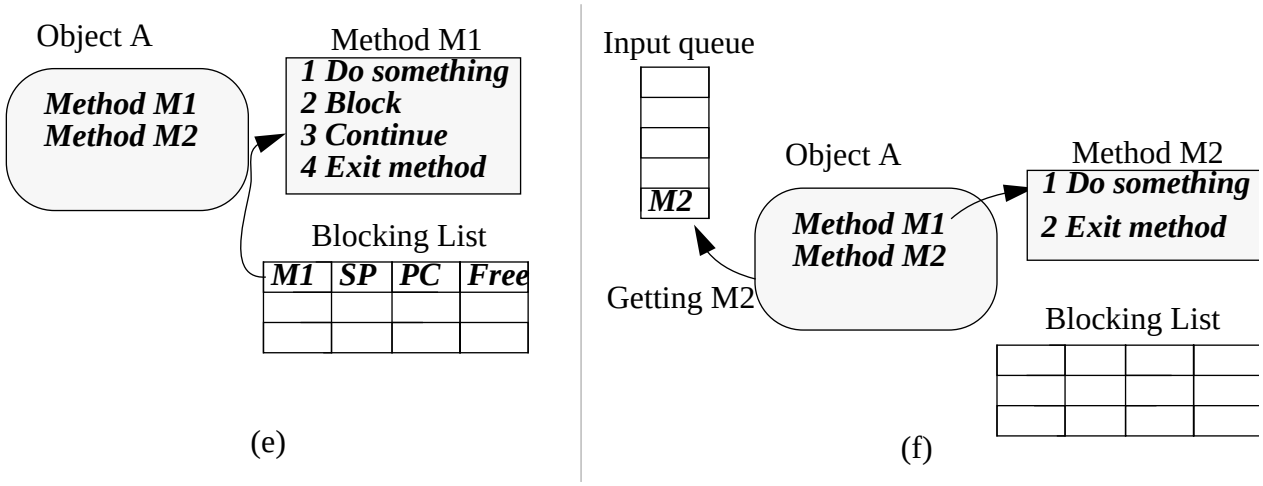


Figure 3. Scheduling within an object

This approach gives a solution for the problem of locking other objects in order to avoid race-condition, because the object referenced by the pointer also is scheduled in this manner as stated above.

One problem remain and that is the existence of static variables. A static variable is an instance variable which is common for all objects within a class. When having several objects of a class having a static instance variable it is likely to have race condition on that particular variable. See the example below, if we makes two instances of the class Foo and these objects have their own thread of control, it would be obvious that a race condition could occur.

This is an example of a static variable:

```
class Foo
{
   public:
      void myMethod() { bar = bar + 1; };
   private:
      static int bar;
};
```

The solution is divided into three stages.

- 1 Define static variables to be common only within a macro object.
- 2 A macro object cannot have static variables.
- 3 Decrease the number of threads of control to be only a single one for the objects of that particular class.

If the input queue for an object is full simply a wait message is sent to the caller of the method, telling that the queue is full and to try later.

6 Real Messages

In order to have a distributed system that uses services from other macro objects than it self, messages must be sent through a network.

The messages are used as an object to object communication, only using the macro object as a “physical” address where the object is resided.

As soon as the message has been delivered to the right macro object, the receiver starts finding the right internal object which should perform the processing of the method and puts the message in the right input queue. See *Figure 4. "Sending a message from one object to another object."*. If the macro doesn't contain the required object or the object cannot process the method because the method does not exists or the method is not available in that security level (See further *Security* on page 12) a rejection message is sent back to the sender.

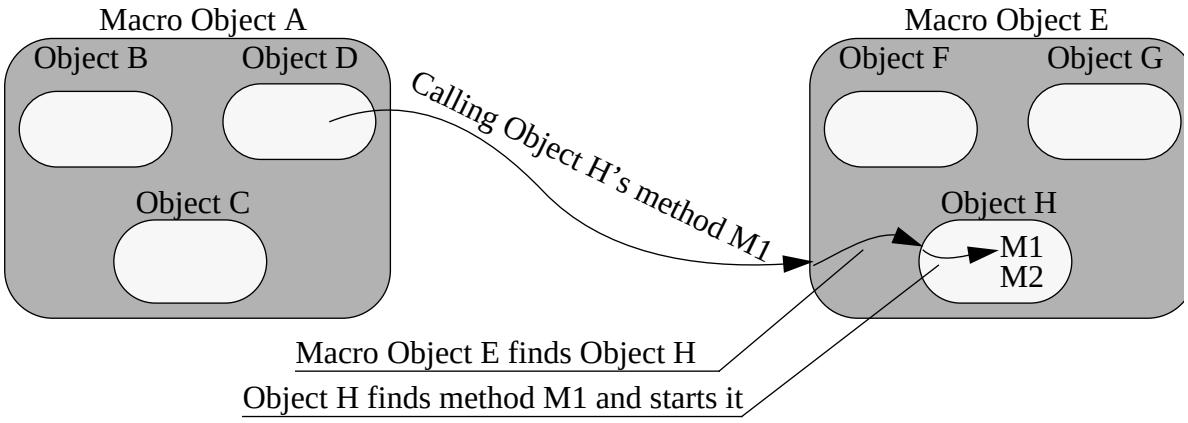


Figure 4. Sending a message from one object to another object.

Then what kind of information should a message include:

- **Macro object class** of the caller
- **Macro object name** of the caller
- **Class name** of the caller
- **Object name** of the caller
- **Macro object name** of the receiver
- **Object name** of the receiver
- **Method name** of the receiver
- **Parameter types** of the input parameters to the receiver
- **Parameter values** to the receiver.
- **Sequence number** of the caller
- **Security code** of the caller

The **Macro object class** is provided to deny access from a macro object that is not allowed to access that certain method (see *Security* on page 12 for further information). The **Macro object name** (of the caller) is required to limit the access for certain instances of a macro class. The **Class name** and **Object name** (of the caller) follows the same ideas as its macro equivalent. The **Macro object name**, **Object name** and **Method name** (of the receiver) is supplied to find the macro object, the object inside the macro object and finally find the required method inside the macro respectively. The **Sequence number** is generated by the caller and is supplied so that when an answer is coming back to the caller with the result of calculating, so that the caller can match the answer to the method which called and mark it as free as in *One Process / (macro) object* on page 3, *Figure 3. "Scheduling within an object"*.

7 Delayed answer

In this kind of domain there are much idle time for a macro object, since it often has to wait for some answer over the network. The network is it self very slow, but hopefully it will be better performance in some years. Today it almost 2 kbits/s, which should serve at least one whole house or even up till one hundred house. If we consider this there will be much idle time just getting a chance to send a message over the network.

In order to increase the magnitude of concurrency an approach is taken towards the concept of futures [Wagner 92]. We will use the same concept, but in a totally different environment and with a totally different goal.

Futures is concerning how to distribute several tasks on several CPU:s, and is because of that very interested in load-balancing, creation times of tasks, in order to do things in parallel. Further they are discussing migration of task when certain CPU:s are more loaded than others.

Our goal with futures is to find something to do while waiting for an answer. It is important to note that in [Wagner 92] the task using a future is possible to run the task in sequential order. But in our domain we cannot execute the task ourselves, we must ask some other task in order to get an answer. We can therefore ignore load-balancing, creation times of tasks, because we cannot move tasks around and there are already running a task “on the other side”. In [Wagner 92] and [Taura 94] the programmer must explicitly define what to be considered as a future.

There exists a concept called Single-assignment [Siviotti 94]. This concept works a little like futures, but we are only possible to write to that variable once, trying to write a second time will cause a run time error. The variable is initially undefined, but as soon as initallized it can be read several times. Trying to read the variable in the uninitalized state will suspend the reader. There are more features in [Siviotti 94], such as atomic transaction not considered in this paper.

Our approach is somewhat between futures and single-assignment, we call it delayed answer. It should be possible to call a method in another object (or even another macro object) and assign the result of that value to a variable. As long as we does not use the variable we will continue to execute our task. But as soon as we are trying to use the variable for computation the task will be blocked until the answer of the previous method call is available. If the answer really is available at the first read there are no tasks needed to be blocked. The developer does not need to explicitly state whether a the answer should be considered as a delayed answer, it is always considered to be a delayed answer. In [Wagner 92] the developer must explicit state where the futures should be.

The scheduler in *One Process / (macro) object* on page 3 is almost perfectly supporting this kind of actions. The task is being blocked when trying to read an uninitalized variable and is marked as free whenever the answer is returned over the network. This works also perfect between object within the same macro object, since it is possible to have a single thread to each object.

Consider the following two classes:

```
class Caller
{
   public:
      int Calling()
      {
         int j, k = remote.Call(100);
         for(int i= 1; i < 100; i++)
            j = j * i;
         j = j + k;
         return j;
      }
```

```
   private:
      Remote remote;
};
class Remote
{
   public:
      int Call(int input)
      {
         int ret = 0;
         for( ; input > 0 ; input--)
            ret = ret + input;
         return ret;
      }
};
```

Here we have a class `Caller` calling in class `Remote` the method `Call`, in traditional computing (such as RPC [Srinivasan 95]) the class `Caller` will be suspended until the method `Call` in `Remote` is finished.

Using our approach the method `Calling` in class `Caller` is not suspended until the line which using the variable `k` (`j = j + k;`). This gives us a possibility to run through the loop in `Calling` meanwhile the method call to method `Call` in class `Remote` is sent and executed. The consequences are that the both loops in `Call` and `Remote` is running in parallel.

In order to keep track of which variable is defined and which are not, only one bit of memory is needed per variable. More, it is only variables that is assigned through methods calls that is needing this extra bit. The bits are placed in a special vector maintained by the compiler.

Each time a variable with the extra bit is read, it must be checked whether the variable is valid or not. One can expect that all this checking will make a lot overhead, but using a little optimization could reduce that over head a lot. Repeated reads will not be checked, only once, for instance.

The *Figure 5. "RPC versus Our Approach"* will show how the two classes will be executed in a normal RPC and in our approach. Note that we assume that communication over the network and the loop in method `Call` will together take longer time than the loop in method `Calling`.

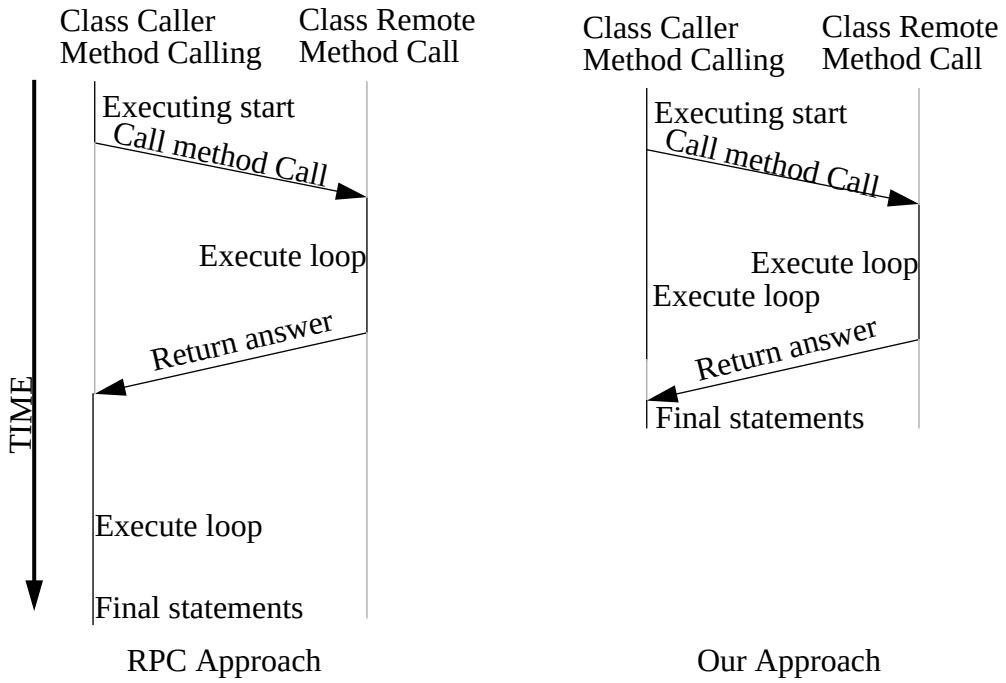
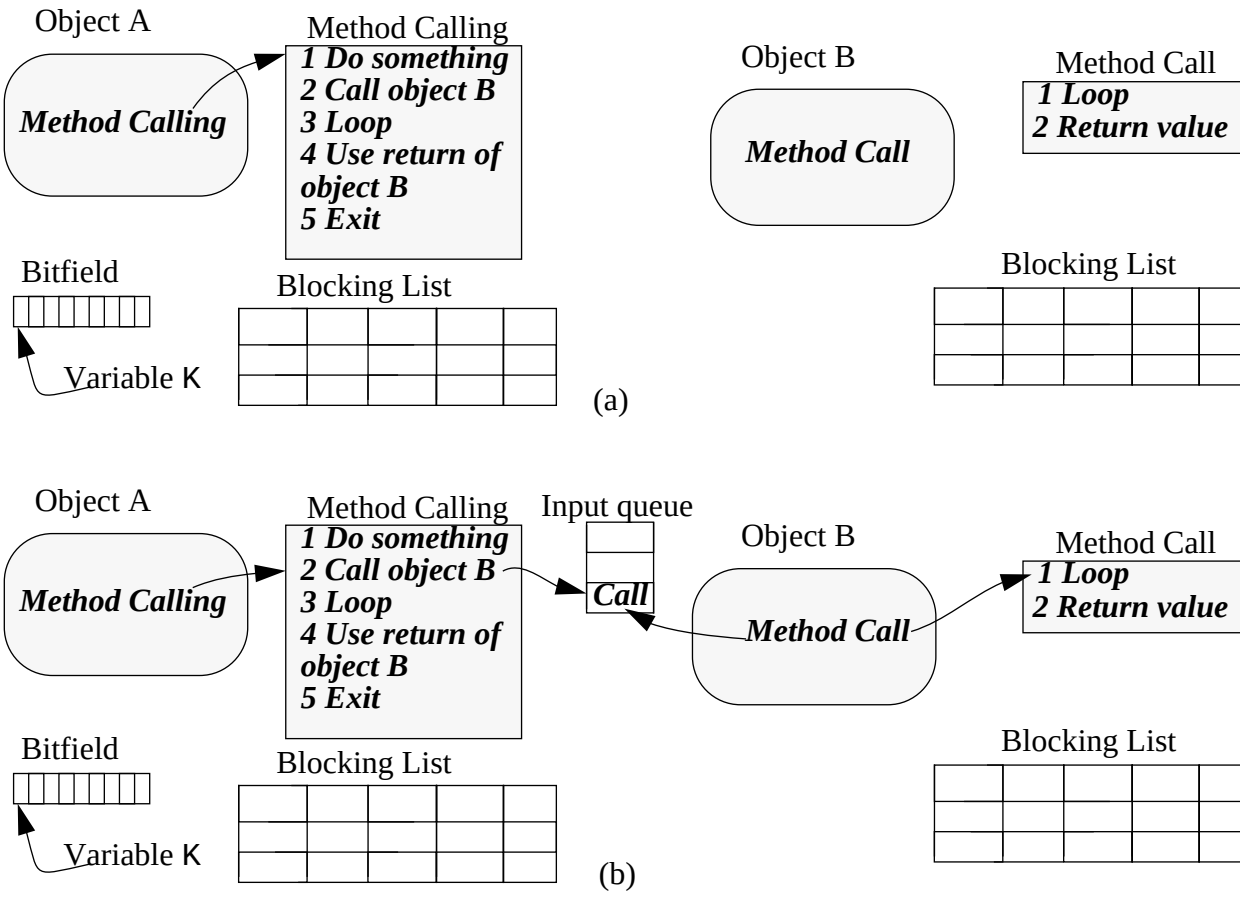


Figure 5. RPC versus Our Approach

How does this fit in to our scheduler (see *One Process / (macro) object* on page 3) is shown in *Figure 6. "Processing a delayed answer"* where object A is an object of class `Caller` and object B is of class `Remote`.



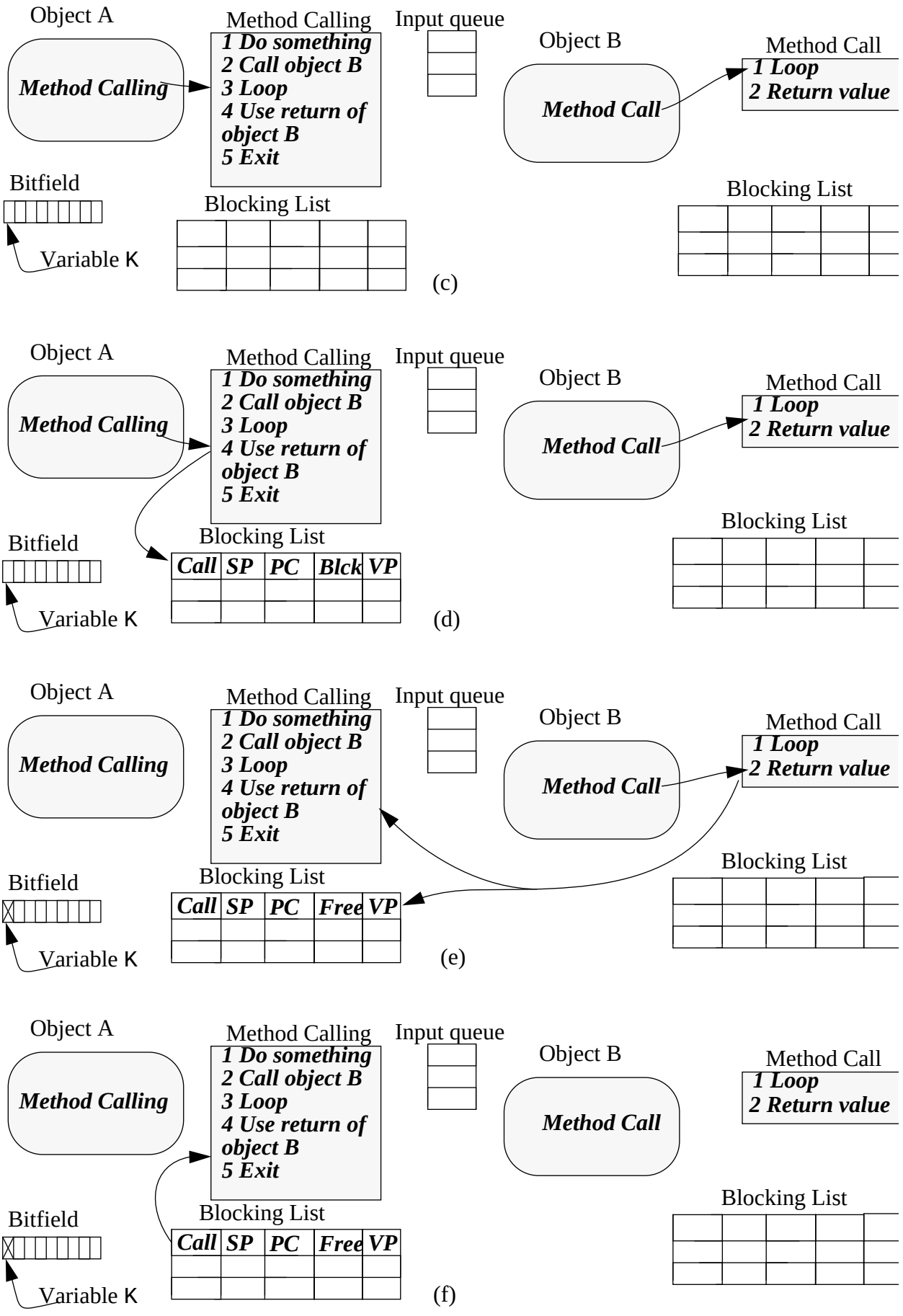


Figure 6. Processing a delayed answer

In (a) *Object A* is starting the method `Calling`. In (b) method `Calling` is calling method `Call` in *object B*. Both loops in method `Calling` and `Call` is running in (c). When method `Calling` is trying to use the variable `k` in (d), which is marked as invalid in the *Bitfield*, the method `Calling` is blocked and put in the *blocking list*, in addition to *Figure 3. "Scheduling within an object"* a slot to save the location of the return variable is stored called *Variable Pointer* (VP). When method `Call` in object B is finished it will return its value to method `Calling` in *object A* in (e). *Object A* will use the *VP* to store the return value and mark the *bitfield* for variable `k` as valid and set method `Call` as *Free*. *Object A* is now able to continue its work as in (f).

In the case of a method is not accessible, the macro object is down or communication is down, there can't possibly be an answer from the receiver of the message. In this case an exception is risen by using the C++ primitive `throw 42`. If no `catch` block is defined or no `try` block is defined the normal "abnormal" program termination occurs [Schildt 94].

8 Security

Since the domain of this paper is a public accessible application, it seems like a good idea to restrict some use for some kind objects. This is because of restricting others getting personal information or even computeral power. In order to achieve that some kind of security levels must be incorporated in the system.

This approach differs from [Strom 83], which is more to be encapsulation.

One other effect of having this ability to reject messages from unwanted objects in the system is to provide different methods for different objects.

For example if Fred's home robot which is going to serve Fred some tea asks the automatic person tracking system where Fred is, the tracking system is telling the home robot in which chair Fred is sitting. On the other hand if Fred's doorbell is asking where Fred is (in order to really use the bell or not, *"There are no use shouting at the deaf"*) the answer to the question from the person tracking system is whether Fred is inside the house or not. See *Figure 7. "Different answer depending on different callers"*.

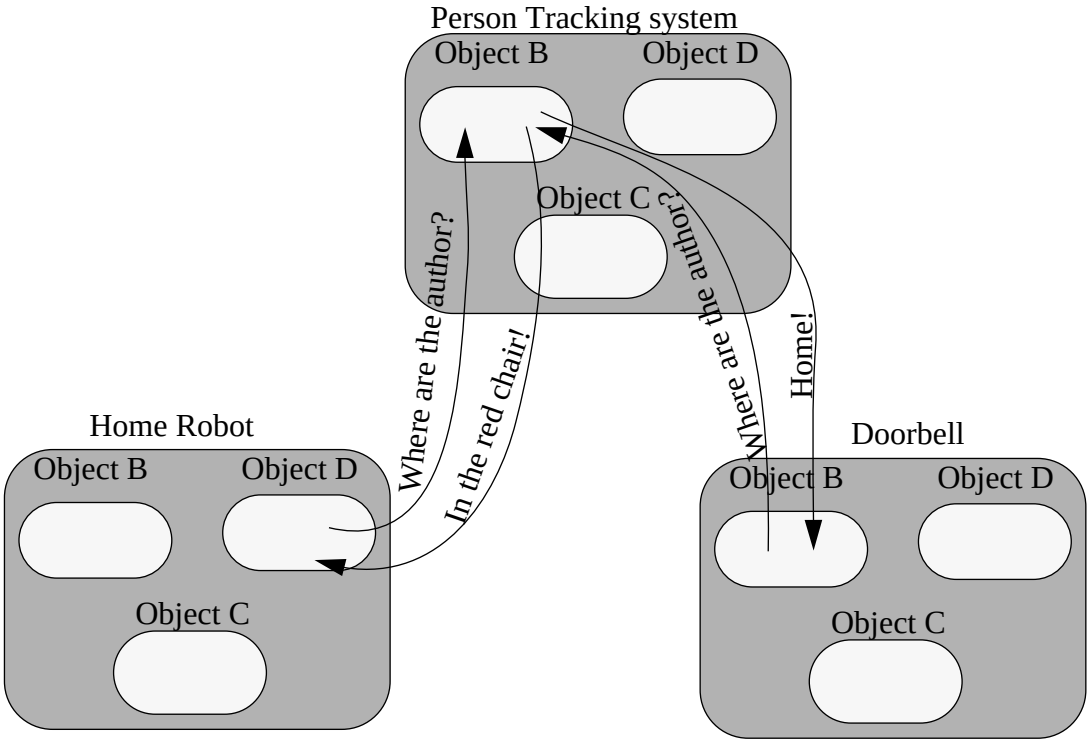


Figure 7. Different answer depending on different callers

This was an example of the same question being asked to the same object but resulting in two different answers.

One other important security thing is that only the manufacturer is able to get some information from the system or calling certain information, i.e. getting the internal state of an object.

An example of the above is that the coffee company are asking your automatic coffee maker if there are enough coffee in the maker to last until next friday, if not the coffee company will send you coffee. Of course you don't want the coffee company's competitor gain access to that kind of information in order to be able to sell their coffee to you.

Simply a normal password would be interesting to have, since this facility will make it possible for the user of the system to gain certain control over the system depending on which person actually wanting the information.

One example is that the rent is paid automatically upon command from you electronic calendar, but in order for the calendar to require access to your account it must provide your personal password.

Finally the identity of the object which call for the service is another approach to restrict the communication between different facilities.

As an example, it is only acceptable for a switch to turn on the light not the dishwasher.

This far we have managed to incorporate all new facilities in our language with out having to define any new language constructs. The compiler is able do deal with the new functionality without the programmer's notice.

To add security a new construct must be defined within the language. There are many possibilities, the one chosen here may not be the most elegant one, but it fulfils the task at this early moment.

The security is divided into four main securities mentioned above, general security levels, manufacturer of the caller, passwords and caller's identity.

The caller's identity is divided into several parts:

- The callers macro class
- The callers name of the macro object
- The class of the calling object
- The callers name of the object

This is defined (in most cases) in the declaration part of the class, see below.

```
class Foo
{
   security:
      organisation = true;
      level = 3;
   public:
      void Bar@firstLevel(void) / org = acme_org;
      void Bar@others(void);
};
```

As we can see in the example the keyword `security` is added. The statements is specifying that this class is granted access to use services that requires the caller to manufactured by the same organization.

The second statement is giving the class ability to reach the security level 3.

There are 16 securities levels ranging from 0 to 15, 0 is the lowest security level and 15 the highest.

The keyword `public`: is used as usual, but after the "/" follows a list of security levels that must be fulfilled to execute that method. In the example the method Bar@firstLevel (more about the @ sign later in this chapter) requires that the calling object has to be within the organization. Several organizations can be defined by writing: `/ org = acme_org, other_org;` meaning that the caller must belong to either *acme* or *other* organization. If the org = is omitted any organization (or none at all) is permitted to use the method.

By specifying yet a slash ("/") the programmer can specify the other securities levels. All the statements between slashes is AND-ed together and all the statements inside the slashes separated with a comma is OR-ed together.

The `level = 4` means that security level four is required to gain access, an additional plus sign after the level number indicates that all higher levels are also sufficient.

By defining `mcrclss` you restrict the callers to belong to a list of macro classes separated with a comma to gain access. An additional plus after each class name indicates that all super classes in the inheritance hierarchy is sufficient to gain access to the method. By specifying a minus sign all sub classes in the inheritance hierarchy is sufficient to gain access to the method. It is also possible by combining two classnames with plus and minus signs to limit the gain for classes in the class hierarchy. In *Figure 8. "Class hierarchy"* we can give class A, B, D and E access to a method by defining `myMethod()/ mcrclss = D-, B+;`

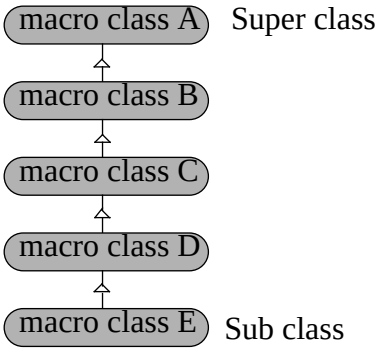


Figure 8. Class hierarchy

In the case of giving certain macro object access the `mcrojb` is used to specify one or several objects.

The class and object calling the other object can be defined in the same manner as with the macro objects using `objclass` and `obj` to specify the class of the objects that may access the method and the object repetitively.

Finally by defining `pswd` one or several passwords must be given by the calling object to gain access to the method.

A summarize of the syntax is given below, "[" and "]" is used to express optional things, "|" to express choices and "..." is indicating that the previous statement can be repeated.

```
method_name(parameter, ...) [/ org = org_name [, ...]] [/ level = number_of_level[+]] [/ mcrclss
= macro_class[+|-][, ...]] [/ mcrobj = macro_object [, ...]] [/ objclss = object_class_name[+|-
] [, ...]] [/ obj = object_name [, ...]] [/ psswd = password [, ...]];
```

To avoid *inheritance anomalies* [Bergmans ??], each method could be redefined in the subclass also without redefining the actual code, thus making it possible to redefine the interface.

The password is dealt in another way, each time a method is going to send a password to another method it is expressed like this in the code:

```
object.method() : password;
```

The password is added after the method call separated with a colon, this leads unfortunately to *inheritance anomalies* and must be accepted at the moment. Further work may come to an solution.

In order to keep all these securities levels and avoiding unauthorised developers writing programs that is accessing classified data, each time the program is compiled the programmer must verify that he or she is authorised to gain that kind of security level by typing some kind of password when compiling.

In code example above there are two methods named `Bar@firstLevel` and `Bar@other`. The `@` sign is used to specify different methods to use depending on which security access the caller has. From the outside there exists only one method called `Bar`, but depending on the calling objects manufacturer the method is selected.

This gives an ability to give different answers to different callers. In the implementation of the class there are two methods implemented called `Bar@firstLevel` and `Bar@other` respectively. Internally an object is able to call any version of the method from any other (version of) method.

In *Figure 9. "Access control"*, a little example is showed.

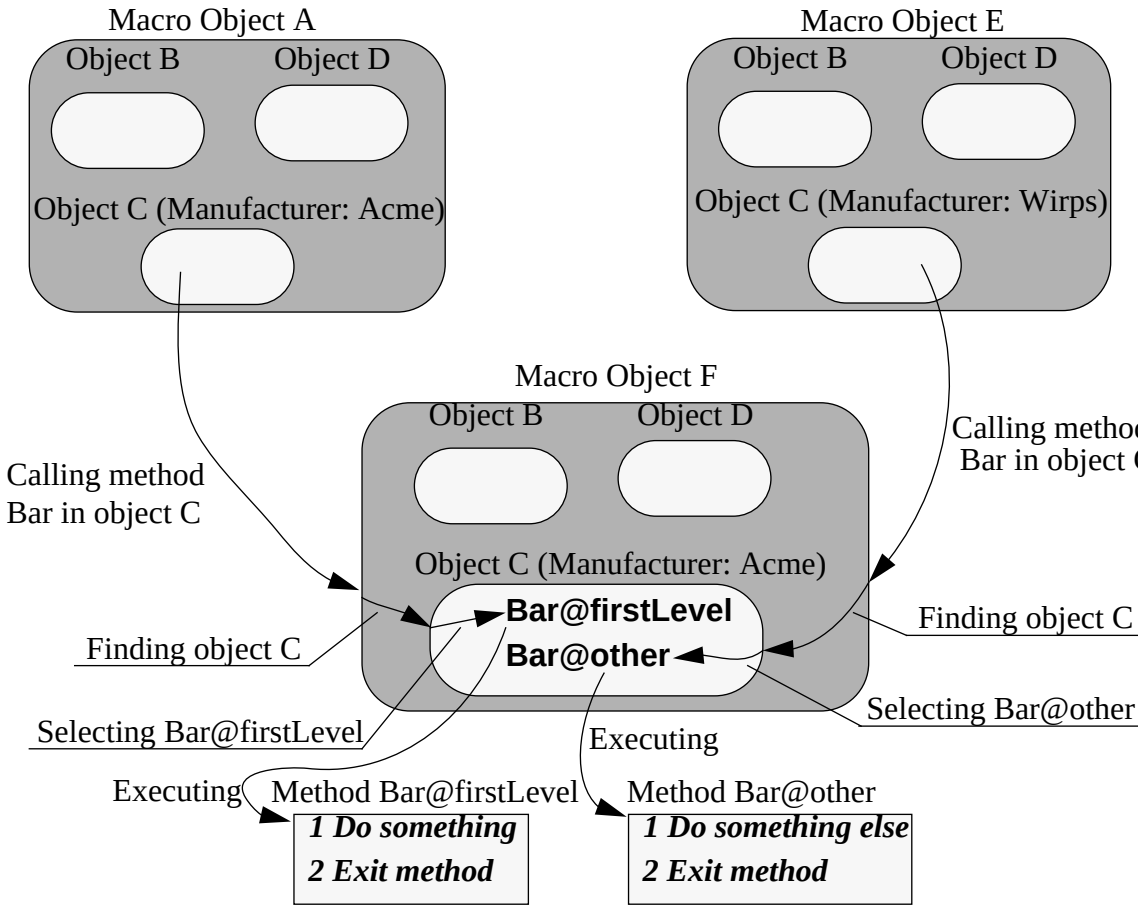


Figure 9. Access control

As we can see, when object C in macro object A calls method Bar in macro object F, method named Bar@firstLevel is executed since this method is accessible because it is same manufacturer of the objects. On the other hand object C in macro object E is not able to execute method Bar@firstLevel when calling Bar, instead method Bar@other is selected for execution.

If no method is accessible by the caller an rejection message will be send to the caller.

It is very important to write the different versions of the methods in the right order. The selection of versions is based upon that the most restricted version is stated above the less restricted versions.

9 Future work and drawbacks

Since this is the first paper we have done in this domain there are a lot of things that will dealt with in the future. One thing is the security, one should be able to change password in real time at this moment you have to change the specification of the method in order to change the password.

Another is when you are about to select a suitable method depending on the security level. At the moment the approach is that the programmer states the versions of the methods in that order that the tests will performs, in other words the system will choose the first applicable method version that not deny access in the order the programmer states the method versions. For example is a version of a method that does not require any kind of restrictions, if it is placed as the first method version, all calls to that method is using that method version. Selecting some kind of closest match would be a nice improvement.

Some kind of multi cast to all objects within a certain class or macro object/ macro class should be nice to have, especially a multi cast to a certain number of objects/classes, then telling that two hundred dishwashers should stop washing for two hours since the energy supplier will have short of energy during to a failure in a generator.

One problem not addressed in this paper is the problem with identification of macro objects, how does old macro objects find newly installed macro objects? For example if each and every light bulb is having a neuron chip (and then is an own macro object). How does my switch know that the light bulb has been replaced by a new one?

These questions above are not considered in this paper.

References

[Bergmans ??] Lodewijk Bergmans, Mehmet Aksit and Ken Wakata, “An Object-Oriented Model for Extensible Concurrent Systems: The Composition-Filters Approach”, University of Tokyo, DATE UNKNOWN.

[Bosch 95] Jan Bosch, “Layered Object Model investigating paradigm extensibility”, University of Karlskrona/Ronneby, 1995.

[Micallef 88] Josephine Micallef, “Encapsulation, Reusability and Extensibility in Object Oriented Programming Languages”, JOOP 88, 1988.

[Schildt 94] Herbert Schildt, “C++ from the Ground Up”, Osborne McGraw-Hill USA, ISBN 0-07-881969-5, 1994.

[Siviotti 94] Paolo A.G. Siviotti, “A Verified Integration of Parallel Programming Paradigms in CC++”, In H.J. Siegel, editor, 8th International Parallel Processing Symposium, pages 44 - 50, IEEE, April 1994.

[Smolka 92] Gert Smolka, “An Oz Primer”, DFKI Oz Documentation Series, Programming Systems Lab, German Research Center for Artificial Intelligence (DFKI), 7 April 1992.

[Srinivasan 95] Raj Srinivasan, “RPC: Remote Proceudre Call Protocol Specification Version 2”, Sun Microsystems USA, August 1995.

[Strom 83] Robert E. Strom and Shaula Yemini, “NIL: An Intergrated Language and System for Distributed Programming”, Sigplan Notices, June 1983.

[Tanenbaum 92] Andrew S. Tanenbaum, “Modern operating Systems”, Eglewood Cliff, NJ, Prentice Hall, 1992.

[Taura 94] Kenjiro Taura, Satoshi Matsuoka and Akinori Yonezawa, “ABCL/f: A Future-Based Polymorphic Typed Concurrent Object-Oriented Language - Its Design and Implementation -”, G. Blelloch, M. Chandy and S. Jagannathan, editors, Proceedings of the DIMACS workshop on Specification of Parallel Algorithms, 1994.

[Wagner 92] David B. Wagner and Brad G. Calder “Portable, Efficient Futures”, Technical report CU-CS-609, University of Colorado, August 1992.

[Wegner 90] Peter Wegner, “Concepts and Paradigms of Object-Oriented Programming, Keynote talk OOPSLA ‘90”, OOPSLA ‘90, 1990.

Developing with Supervisor Objects

A Method to Increase Traceability in Software Development

Martin Walfisz
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: martinw@pt.hk-r.se
www: http://www.pt.hk-r.se/~martinw

Abstract
Traditionally Object Oriented development methods ensure structuredness and encapsulation of objects. This is said to lead to higher reusability, portability and maintainability. However, the aspects of tracing system requirements throughout the development phases are not easily incorporated in todays methods. In this paper we introduce the notion of Supervisor Objects, a design principle and development method which automatically enforces traceability through all phases of software development. Using Supervisor Objects leads to better maintainability and control of modules developed in a system. A Supervisor Object acts as a manager and interface controller over a set of cooperating objects, together forming a module. By using Supervisor Objects we can also increase the understandability and readability of developed code modules.

Keywords: Trace, Traceability, Requirements Traceability, Object Orientation, Object-Oriented Requirements.

1 Introduction

Todays Object Oriented languages and development methods are angled at producing well structured and encapsulated systems. They deal with reusability, portability, maintainability, etc. by adding structure and encapsulation to the objects within the system. Traditionally the development methods start by composing a requirement specification on which the system is based. The requirement specification can be seen as a document containing a complete description of the systems external behaviour [Davis 93] and it is the single most important document during development since it is the basis for all further progress. In order to increase maintainability it is very important to allow for traceability between the requirements and the developed system entities. However, a study [Lindvall 94] shows that even when using modern and generally accepted object oriented development methods the traceability of the requirements throughout the system can easily be lost.

1.1 Traceability

During the development the requirements are translated into a design which represents the needed system. The design is then translated into source code which is compiled into an executable system. When we have a clear connection between the requirements and the developed source code we say that we have good horizontal traceability. Each connection between one requirement and the piece of source code that imple-

ments that requirement is called a horizontal traceability link. Vertical traceability is when we have a clear view of what dependencies and connections exist between entities in the same phase, for example between two requirements or between two source code modules. In this paper we mainly investigate horizontal traceability, hereafter only called traceability unless explicitly stated otherwise. The t raceability links are very helpful when performing maintenance since they increase the possibilities to see dependencies and understanding of the system structure.

1.2 Problems

The currently available development methods do not automatically help in tracing requirements from the requirement specification down to the developed source code. If we do not emphasize on traceability when developing software

- we decrease maintainability by not having clear dependencies between parts of the system,
- we decrease the possibility to understand the system since connections are not clearly identified and
- we increase the risk of losing requirements since we have no easy way of directly controlling what requirements have been implemented.

Another problem with traceability is that the requirement specification information is often distributed all over the system, hence making it very difficult to directly trace single requirements [Finkelstein 91], this also implies that there is no one-to-one relationship between requirements and code, i.e. the implementation of requirements is often divided between many code blocks and one piece of code can implement several requirements [Tilbury 89, Kelley 90]. Further, the tracing of non-functional requirements is traditionally not considered in todays development methods [Lindvall 94], making it very hard, if not impossible, to trace the implementation of such requirements.

1.3 This Paper

What we do in this paper is to propose a design principle that enforces traceability between the requirement specification and the developed code while still using almost any object oriented development method and programming language.

This paper is organized as follows, in section 2 we define and explain the concept of Supervisor Objects. Section 3 explains how to develop with Supervisor Objects and in section 4 we conclude the paper. Appendix A contains an example of how to implement part of a small system using Supervisor Objects.

2 Introducing Supervisor Objects

One approach to increase traceability is to use a development method which uses the same paradigm throughout all development phases, i.e. any concept found in one phase can directly be mapped to a similar concept in the successive phase. This is called seamless development [Jacobson et. al. 92]. The problem with this solution is that currently there exist no development methods where you use entirely the same paradigm throughout all phases. Also, since the paradigm changes between different application domains the development method must be able to incorporate the new semantics. Some research has been done in let-

ting programming languages be extendable with new concepts ([Bosch 95]), but currently the industry uses traditional object-oriented programming languages such as C++ [Stroustrup 86], SmallTalk, etc. which are not capable of incorporating new concepts that fit the development domain.

Another approach for increasing traceability is to use the concepts currently available in programming languages and extend the ways in which they are used. We propose the concept of Supervisor Objects which can be used to increase traceability links with almost any object oriented programming language and design method.

A Supervisor Object is an active object which acts as a manager or controller for a set of objects, or a module[1]. It can be viewed as a person controlling a set of devices, using them to achieve a task. The Supervisor Object fulfils a set of requirements put on the module it supervises. Any other object can ask the Supervisor Object to fulfil a certain requirement which the module's requirement specification states it is capable of achieving. Figure 1 shows the conceptual view of how Supervisor Objects interact with modules.

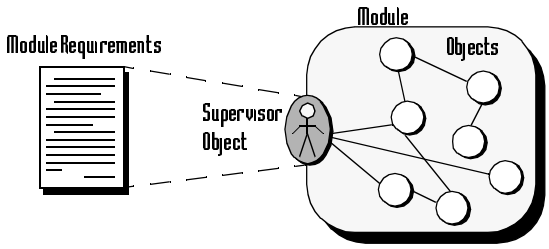


Figure 1. The Module's requirements are mirrored in the Supervisor Object.

2.1 Properties

Each Supervisor Object has a few general properties that help the development and maintenance of software system.

- It acts as a **module interface** since all messages[2] must pass through the Supervisor Object. Thus it intercepts all incoming and outgoing messages to and from the module.
- It **actively monitors and controls** the module by having the possibility to examine and intercept all incoming and outgoing messages and by being able to independently interact with the objects in the module.
- It **enforces traceability links** since all module requirements must be represented in the Supervisor Object, thus automatically creating connections between the requirements, design and produced code.

A Supervisor Object also increases understandability since it contains all information about what the module can perform, i.e. a maintainer of a module only has to read and examine the Supervisor Object in order to understand what requirements the module fulfils.

1. We define a module as any set of objects or sub-modules that cooperate to perform a task. An application is defined as a module containing several sub-modules cooperating to fulfil the complete systems requirements.
2. We define a message as a call to or a reply from an object within a module.

2.2 Module Interface

The Supervisor Object acts as an interface to the module. Any calls to the module must go through the Supervisor Object and it is the Supervisor Object that decides whether the call is valid or not. This allows for Supervisor Objects allowing different messages in different system states.

Dynamic Behaviour. One problem associated with object oriented design methods is that there is not way to easy model dynamically changing requirements, i.e. there is no way to model change of the requirements during execution to conform to different states of the system. However, by designing several Supervisor Objects to interface a module we can dynamically bind them during run-time and thus changing the way the requirements are handled dynamically.

Another problem which is solved by the dynamic behaviour of Supervisor Objects is that we easily can restrict the way modules are accessed during execution. For example a system which has several authorization levels may use Supervisor Objects to realize the authorization levels depending on who is using the system, i.e. a low-authority user can only access the module through a low-authority Supervisor Object, while a high-authority user will access the module through a high-authority Supervisor Object. Thus, code which normally is entered into all objects to restrict access can instead be put in the Supervisor Object, hence it is not spread over several objects. Figure 2 shows an example of how the interface of a module dynamically can be modified by replacing the Supervisor Object.

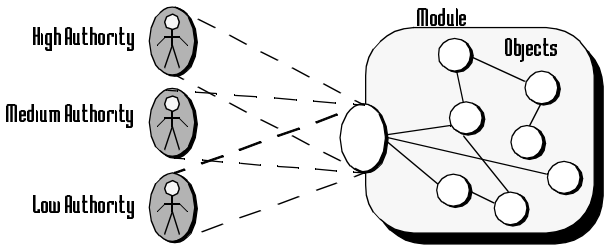


Figure 2. Dynamically selecting Supervisor Object depending on the systems state.

User Interface Interaction. For modules that interact with a user interface the input and output from the user interface must go through the Supervisor Object (see Figure 3). This gives the advantage that the coupling between the user interface and the system is very lose, thus enabling easy modification without the need to resolve user interface dependencies. By combining the user interface with the dynamic behaviour of Supervisor Objects we can very easy change the way the user interface interacts with the system, i.e. different users can interface different Supervisor Objects.

By putting the user interface as a separate entity outside the system or module, we help resolve the design issue of where the user interface should be connected to the system. Our view is that the system acts as a black box to the user, and the user interface is the means for the user to communicate his desires to the system, i.e. the user interface is a communications media, placed, conceptually, between the user and the system.

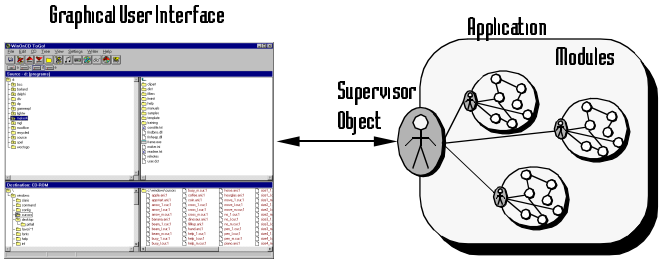


Figure 3. A Graphical User Interface accessing the Application through the Supervisor Object.

2.3 Actively Monitor and Control the Module

A Supervisor Object actively monitors the behaviour of the objects within the module. It is the Supervisor Objects responsibility to make sure that the requirements put on the module always can be fulfilled, therefore it has to be able access the objects within the module and constantly ensure that they can help in fulfilling the requirements.

Functional requirements. Functional requirements are handled by the Supervisor Object's method interface, i.e. each functional requirement is designed and implemented as a method in the Supervisor Object.

Non-Functional requirements. Non-functional requirements can automatically be ensured by the Supervisor Object since it actively can examine objects within a module.

- Performance requirements. Different performance requirements put on the module can be monitored and controlled. For example if a communication module has a requirement that it must be able to communicate with a speed of 9600 bps or more, the Supervisor Object can monitor the connection and take appropriate actions when the speed is too slow.
- Real-Time requirements. A Supervisor Object can include code for controlling real-time requirements of a module. For example a module can have a requirement that it must perform a certain task within 20 ms. The Supervisor Object can then monitor any calls to that method and perform different actions depending on success or failure of the time to process the call.
- Concurrency requirements. By controlling the concurrency level within a module a Supervisor Object can conform to concurrency requirements such as only allowing one call to a module at a time or ensuring the atomicity of critical regions.

Event Handling. Since the Supervisor Object can act as an active object, i.e. it can be a process in it self, it can easily perform different actions depending on what happens in the system. For example, the Supervisor Object can implement a timer event which triggers an object call every 10 seconds in order to help fulfil a requirement.

Testing. During testing of modules it is very easy to integrate black-box module tests by implementing the tests in the Supervisor Object. A Test Supervisor Object can dynamically replace a normal Supervisor Object when, for example, the system is in debugging mode. A Supervisor Object can also be implemented to constantly perform tests of various kinds during execution, this makes it possible to early notice errors in a system and for example perform graceful degradation.

2.4 Enforce Traceability Links

By developing with Supervisor Objects we enforce the creation of traceability links between the requirements, the system design and the developed code. These links can be used during development and maintenance to increase the understandability of modules in the system, hence allowing for faster development and maintenance. We also minimize the risk of losing requirements since we are able to directly map requirements to a specific piece of code. In Figure 4 we see how different parts of the requirements are linked to Supervisor Objects in the system.

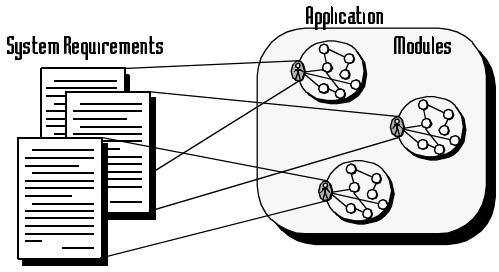


Figure 4. The Direct Mapping of System Requirements to Supervisor Objects in the Application.

3 Using Supervisor Objects

Even if the concept of Supervisor Objects is somewhat trivial, the actual use and implementation requires some effort. We propose to use the following phases when developing with Supervisor Objects.

Identify and Specify Requirements. During this phase all requirements which the system is to be built upon must be elicited and specified. The requirements must be decomposed into small enough entities to ensure the possibility of directly connecting them with a module. They must also be uniquely identifiable in order to be able to make references to them. For more information on how to write well-structured requirement specifications we refer to [Davis93].

Analyse and Identify System Module Requirements. After the requirements have been specified we must analyse and decompose the system into modules and sub-modules. Each module will have its own requirement specification that is connected to the general system requirements. Before this phase is over we must make sure that all general system requirements have been covered in the module requirements.

Identify and Design Supervisor Objects. Each module identified must be assigned a Supervisor Object and each Supervisor Object must be designed to contain all the requirements which the module is supposed to implement. The requirements are designed as methods in the Supervisor Object so that they can easily be traced. We recommend to name the requirement methods similar to the requirements they denote in order to quickly be able to map the requirements specification against a Supervisor Object.

If a module uses several Supervisor Objects dynamically it is important to make sure that even if a requirement is not valid in a certain state, i.e. a particular Supervisor Object does not support the requirement, it should still be implemented as an empty method. This ensures that all requirements can always be traced in the Supervisor Object.

A recommendation is to always write a short comment in each requirement method that exactly specifies what requirement, in the module requirement specification, the method implements.

Design and Implement Modules. The design and implementation of modules and objects within modules is done in the same way as normal object oriented development. However, it is important to remember that, together, the objects in the module must, be able to fulfil the functional and non-functional requirements put on the module.

Implement Supervisor Objects. The implementation of Supervisor Objects is performed after the design of the modules has been done so that the Supervisor Objects know how they can use the module to achieve the requirements. Since the Supervisor Object is responsible of fulfilling the module requirements it is also responsible for ensuring that the complete module has been instantiated and has been initialized for use.

Check Traceability Links. After the modules and Supervisor Objects have been implemented it is, of course, important to verify that all requirements can be traced throughout the development phases. The easiest way to do this is to make sure that every requirement in the requirement specification can be directly mapped to an implemented method in the Supervisor Objects.

3.1 Language Support for Supervisor Objects

In order to be able to fully use the concept of Supervisor Objects the programming language in which the objects are implemented should support a few concepts:

- Dynamic Binding. To allow for the dynamic behaviour of Supervisor Objects the programming language must support dynamic binding, i.e. we must be able to replace objects in the system during execution. Preferably the language also supports refinement of methods in objects during execution allowing for total dynamic Supervisor Object modifications.
- Multi-processesing. In order to be able to actively control and monitor objects within a module each Supervisor Object must be implemented as a separate thread of execution [Tanenbaum 92], thus, the operating system and programming language in which the Supervisor Object operates must allow for multi-threading. Of course, many aspects of Supervisor Objects can be implemented without the need of a multi-threaded system.

4 Conclusion

In this paper we have proposed a development method and design principle to help avoid the rise of problems caused by lack of requirements traceability in software projects. The method can be used together with almost any other object oriented development methods and programming languages since it uses existing semantics.

We have shown how Supervisor Objects can be used to increase traceability, increase module control and add possibilities to implement non-functional requirements. The concept of Supervisor Objects is very simple but we believe it to be a powerful method to easily improve software development.

Some problems such as how to decompose and find requirements are not addressed in this paper. However, we think that the emphasis which is put on finding requirements when developing with Supervisor Objects simplifies understandability of requirements, and even though the notion of Supervisor Objects is mostly common sense we believe that it gives a nice conceptual model of how to implement and decompose parts of software applications to allow for requirements traceability.

Acknowledgements

This paper was written during a course in Advanced Object Oriented Concepts held by Jan Bosch. We would like to thank him for his thoughts and comments on the ideas expressed in this paper and for his efforts to make it a very interesting course.

Some of the ideas in this paper have also been conceived during discussions with members of the MPT95 class at the University of Karlskrona/Ronneby.

References

[Bosch 95] J. Bosch, “Layered Object Model, investigating paradigm extensibility”, 1995, Doctorial Thesis, Department of Computer Science, Lund University.

[Davis 93] A. M. Davis, “Software Requirements. Objects, Functions and States”, 1993, Prentice Hall, ISBN 0-13-562174-7.

[Finkelstein 91] A. C. W. Finkelstein, “Tracing back from requirements”, 1991, IEE Colloquium on Tools and Techniques for maintaining traceability during design.

[Jacobson et. al. 92] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, “Object-Oriented Software Engineering. A Use Case Driven Approach”, 1992, Addison-Wesley, ISBN 0-201-54435-0.

[Kelley 90] C. Kelley, “Does it fit the bill?” Systems International, 1990.

[Lindvall 94] M. Lindvall, “A Study of Traceability in Object-Oriented Systems Development”, 1994, Licentiate Thesis, Linköping Studies in Science and Technology, ISBN 91-7871-456-7.

[Stroustrup 86] B. Stroustrup, “The C++ Programming Language”, 1986, Addison Wesley.

[Tanenbaum 92] A. S. Tanenbaum, “Modern Operating Systems”, 1992, Prentice Hall, ISBN 0-13-595752-4.

[Tilbury 89] A. J. M. Tilbury, “Enabling software traceability”, 1989, IEE Colloquium on The Application of Computer Aided Software Engineering Tools.

Appendix A

This appendix shows a simple example of how Supervisor Objects can be implemented in the software for a fire alarm system. The fire alarm system consists of three parts: a control panel, where the fire alarm system can be turned on and off, a fire sensor which detects fire and an alarm bell which sounds when the system detects fire.

Identify and Specify Requirements. We have identified three general system requirements:

SYSTEM REQ 1. By toggling the switch on the control panel the system is turned on and off.

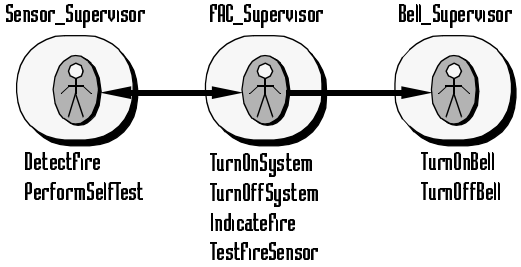
SYSTEM REQ 2. If the sensor detects fire the system must turn on the alarm sound.

SYSTEM REQ 3. The system must run a self-test on the Fire Sensor every hour to make sure that it works.

Analyse and Identify System Module Requirements. The system consists of three modules:

- The Fire Alarm Controller (FAC). This module interacts with the control panel (User Interface) to either turn the system on or off. It has three requirements:
  REQ 1. If the switch on the control panel is toggled it must turn on or off the fire alarm system. (Connected to general requirement 1.)
  REQ 2. If the Fire Sensor indicates fire it must turn on the alarm bell. (Connected to general requirement 2.)
  REQ 3. Every hour the FAC must perform a self-test on the Fire Sensor. (Connected to general requirement 3.)
- The Fire Sensor. This module detects if there is a fire. It has two requirements:
  REQ 1. If the Fire Sensor detects fire it must send a message to the FAC.
  REQ 2. If the Fire Sensor receives a SELF-TEST message it must perform a self-test and return the result to the FAC.
- The Alarm Bell. This module is turned on and off by the FAC. It has two requirements:
  REQ 1. If an ON message arrives to the module the alarm bell must sound.
  REQ 2. If an OFF message arrives to the module the alarm bell must be turned off.

Identify and Design Supervisor Objects. Since we have three modules we also have three Supervisor Objects. Each object contains a a number of method representing the requirements put on their respective modules.



Design and Implement Modules. We have identified three modules:

1. The Fire Alarm Controller module does not consist of objects, it however has interaction with the control panel which can be seen as the user interface.
2. The Fire Sensor consists of a Sensor object connected to a physical sensor which indicates fire.
3. The Alarm Bell consists of a Bell object connected to a physical bell that can be turned on or off.

The design and implementation of the modules is not shown since they are not needed to exemplify the use of Supervisor Objects.

Implement Supervisor Objects. All of the below examples use a simple form of pseudo-code to explain the implementation of the objects. The // characters indicate a comment.

```
object FAC_Supervisor
    method TurnOnSystem                    // FAC requirement 1
        // do something
    end method

    method TurnOffSystem                   // FAC requirement 1
        // do something
        Bell_Supervisor->TurnOffBell
    end method

    method IndicateFire                    // FAC requirement 2
        Bell_Supervisor->TurnOnBell
    end method

    method TestFireSensor repeat every hour    // FAC requirement 3
        if Sensor_Supervisor->PerformSelfTest fails
            // do something
    end method
end object

object Sensor_Supervisor
    method DetectFire repeat every second      // Fire Sensor requirement 1
        if Sensor->IndicateFire
            FAC_Supervisor->IndicateFire
    end method


    method PerformSelfTest                 // Fire Sensor requirement 2
        return Sensor->SelfTest
    end method
end object

object Bell_Supervisor
    method TurnOnBell                      // Alarm Bell requirement 1
        Bell->On
    end method

    method TurnOffBell                     // Alarm Bell requirement 2
        Bell->Off
    end method
end object
```

Check Traceability Links. After we have finished design and implementation we check the traceability links by examining the code for the Supervisor Objects. If each requirement for every module has a method linked to it we have established the traceability links. In our example we see that every requirement has a connection.

Designing reusable software for home system networks

Mats Bäckström
University of Karlskrona/Ronneby
Programvaruingenjörslinjen
S-372 25 Ronneby, Sweden
e-mail: mpt95mba@pt.hk-r.se

Abstract
In the latest years we have seen how we slowly turns from an industry based society to an information based society. Applications within the energy distribution domain will be no exception. The energy infrastructure, today only used for distributing electricity, will be used for these new customer services. Devices in our houses and companies will be connected to local home system networks, where devices communicate with each other and the outside world. These new services will be highly dependent of well defined and specialised objects and groups of cooperating objects. The intention with the report is to show that object-orientation and knowledge of existing design solutions can be to great help when designing software for home system networks.

Keywords: design patterns, frameworks, reuse.

1 About this report

Some of the ideas in this report are only treated in a technical perspective without knowing the needs of the end customers. This means that there may occur integrity conflicts between the individual customer and the producer or distributer of electricity. In a commercial application we must of course take the individual customers integrity with greatest care.

Section 2 will introduce the Home System Networks. We will use a concrete example all through the report to bring object-orientation and design patterns into the home system domain. In Section 3, 4, 5 and 6 we use patterns designing an energy controller. The design patterns used are abstract factory, proxy and the reactor pattern.

2 Home System Networks

Electricity is an amazing product. By pressing a button you have instant access to light, heat, power for your daily habits and electrical equipment. But we must admit that we use the electricity in a very old-fashioned way, many times just like people did in the beginning of the century. Home systems makes it possible for electrical devices to communicate with each other. New remote control of devices saves money for both the customer and the power supplier. All this is possible because techniques allowing data communication through the ordinary power lines has been cheaper to produce and distribute.

2.1 Example of home system applications

- **Distributed Meter Controlling (DMC).** The power supplier reads and controls meter values in industries and houses remotely. There is no need to send personnel to your house for controlling and reading the meters any more. This saves money and facilitates exact meter reading.

- Your home can for a small cost be connected to an alarm central which guards your home for fires and theft. The fire department will automatically be contacted if your house is on fire. The local security company will directly be contacted when there is a burglary in your house etc. By using the existing power lines, no new infrastructure is needed for installing the fire detection/alarm devices etc.
- **Hard To Steal Devices (HTSD).** You can program a particular device to work on the power lines in your own house but on no other power lines. If someone steals your device it will be useless because it doesn't work on foreign unauthorized power lines. Further, the device can be traced to an exact location through the electrical power lines. Let us call these devices hard to steal devices.

2.2 Home systems and object-orientation

Applications for the home system network can take advantage of many techniques used in object-orientation. Concepts like encapsulation, inheritance and dynamic binding should be to great help when building and reusing components for the home system network.

Physical devices like sensors and heating devices etc. can easily be modelled as objects. Because of the great number of objects that will exist at the same time, every object should be as simple as possible with a clear and specific task. The objects should be well defined, encapsulated and communicate in a private manner with each other. The objects can easily be changed and groups of objects can be reused in different kinds of applications. These systems will unfortunately be very complex because of its distributed structure.

The building modules used in a home system must have an abstract design, so that we don't lock our self to a certain type of device or manufacturer. Devices and components with new and better functionality will come up all the time. The home system network must be robust to the frequent number of changes that is typical for a house configuration. This is done with transparent and open-ended solutions. Design patterns and frameworks seems to be promising tools when designing such systems. In this report we restrict our self to use design patterns.

A design pattern is a general description of how to build a certain part in a system. The description is abstract because it concerns abstract design, not a particular design. This leads to reusable components that can be used over and over again. Many of the design patterns are related to each other and uses inheritance hierarchies to build an abstract structure that can easily be reused.

Design patterns provides a common vocabulary for designers to communicate, document and explore abstract design that works in many domains. It also reduces system complexity when complicated design structures are encapsulated into simpler ones. Ones you have understood the structure and learned how to use one particular design pattern its easy to learn new ones.

3 Designing an energy controller using design patterns

3.1 Background

In the future many electrical devices at your home can communicate with each other. Before we start to discuss what an energy controller is, let us give a very short perspective of the domain, a simple home system network. Look at the house in figure 1.

Figure 1. Home system network

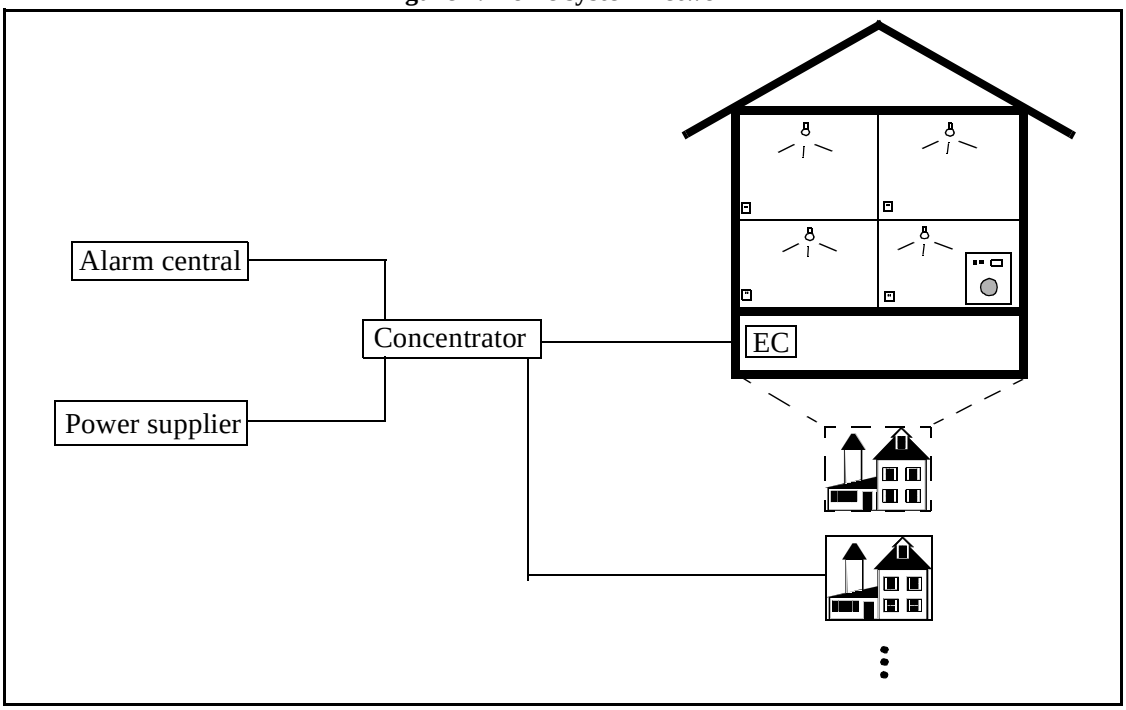


Figure 1. shows how a number of houses are connected to a concentrator. The main functionality of the concentrator is to route services between houses and the power supplier back and fourth. The EC, an Energy Controller can symbolise the intelligence within a house. The EC has communication interfaces to the devices used in the house and the interface to the concentrator, it is also responsible for different house configurations. Lets list some characteristics of this house:

- When a device is used it contacts the energy controller. You can there fore exactly see what devices that are used for the particular moment, or what was used yesterday etc.
- The power supplier have a contract with you to turn off devices with high electricity consumption (for instance, heating devices) when you are not at home. This will save money for the supplier because he can plan much better to reduce peak loads in the power lines. You on the other hand will get a lower tariff price.
- No need for external timers to turn on/off devices in the house. Instead the energy controller can be programmed to turn on/off any wall socket at any time.
- All equipment in this house will be unusable on foreign unauthorized power lines. The devices follow the Hard To Steal Device standard. This is done by sending device codes to the energy controller within frequent time periods.
- Event programmable. For instance, when you enter room r_5 the wall socket ws_3 should change its state to ON.
- Automatic fire and theft alarm.

- Lights will cooperate with the daylight. The ef fect on the lights will be reduced when the light outdoors is increasing. This saves energy.

3.2 The energy controller

The home system seems to need some kind of controlling device, which is the energy controller (EC). Think of the EC as a simple computer with different communication interfaces. This report is written with the scenario that an EC is installed in all homes in the future. Now, what requirement must an EC meet? Lets restrict our self to only find the most important requirements on a very abstract level:

1. Keep track of devices in your house. Adding, updating and deleting different kinds of devices must be very general because your particular house configuration is varied with new devices all the time.
2. Device control.
3. Provide services for the outside world.

3.3 Design patterns

In the following sections we will try to discuss some solutions on the listed requirements in section 3.2 using some existing and well known design patterns. Again, the intention with this report is not to present a full scale design of an EC, but rather show that object-orientation and knowledge of existing design can be to great help when designing home systems. By change our approach to software construction away from code-based and move towards model-based development, our software becomes much more reusable.

4 Using the abstract factory pattern

To meet requirement 1, the EC should be independent of how the devices in your house are created, composed and represented. This is because different kind of devices are being developed by different manufactures using different communication interfaces. For example, the smoke sensors in your house are made by *companyA* but you have also upgraded your house with smaller and cheaper smoke sensors made by *companyB*, and you still want both of them to work together with your EC.

The EC must absolutely not hard-code the functionality to work against one particular manufacturer. Instead you should wait with selection of a certain device until run-time so that the EC can work with devices from any manufacturer.

One useful design pattern that could be used to solve the previously scenario is the **abstract factory**. In figure 2 we declare an abstract factory called **`DeviceFactory`** that is independent of the manufacturer. The **`DeviceFactory`** can create sensors and switches made by any manufacturer.

The manufacturer dependency are moved into the subclasses, one factory for Tohmson and one for Echelon which are two big developers for home system equipment.

The **`FrameWorkCtrl`** together with the **`DeviceFactory`** forms an abstract engine that will work in exactly the same way for all sub factories and independent of the manufacturer.

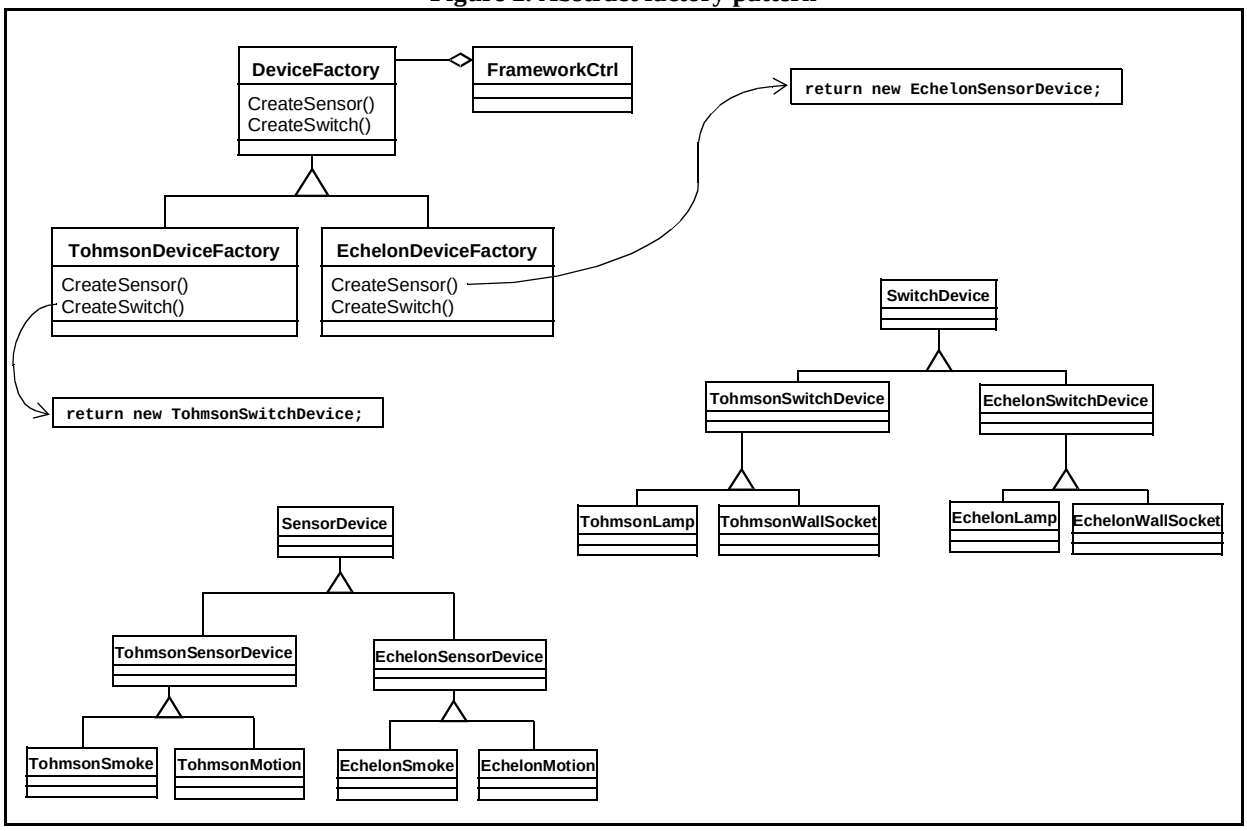
The subclasses uses sensor devices (smoke, temperature etc.) and simple switch devices (wall sockets, lamps, locks etc.) to illustrate the manufacturer dependency. If we want to create a manufacturer dependent sensor like an Echelon sensor then we call the method **`CreateSensor`** using the **`EchelonDeviceFactory`**. The **`CreateSensor`** method will immediately return an instance of an Echelon sensor. The following C++ code can be used to illustrate the scenario:

```
DeviceFactory* factory = new EchelonDeviceFactory;
factory->CreateSensor();
```

```
...
while (...)
   factory->InitializeSensor();
...
while (...)
   factory->InitializeSwitch();
```

In the example the factory type is chosen by using the is-a relationship between an **`EchelonDeviceFactory`** and the **`DeviceFactory`**.

Figure 2. Abstract factory pattern

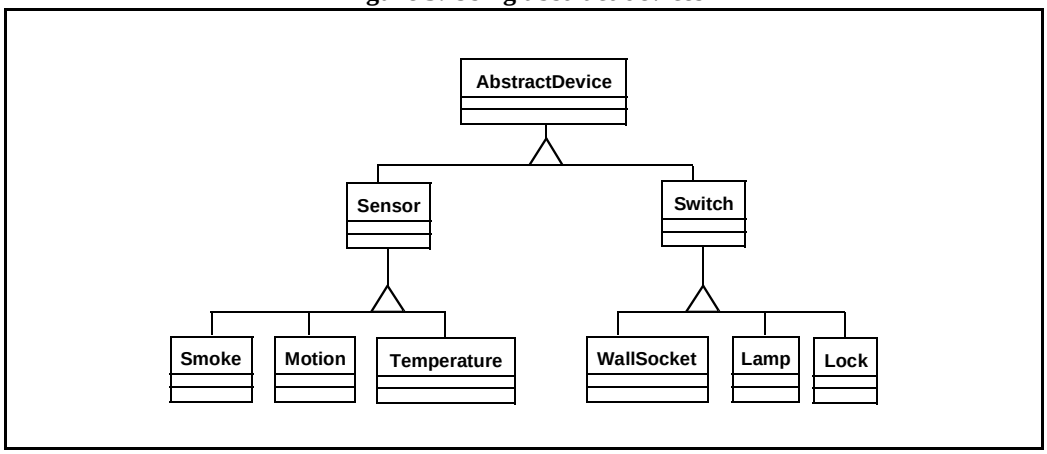


This seems to be fine, but we still have a problem. We have hard-coded the device type into our system by writing:

```
...
factory->CreateSensor();        // Hard-code a device to be a sensor device
...
```

To increase reusability we should assume that working with devices is almost the same independent of what device we actually use. This leads us to the **`AbstractDevice`** that could be any device. In figure 3 we creates an inheritance graph with the most general device type in the top and the least general device type in the bottom. The higher up in this graph we work when designing the EC, the more reusable design. In fact, this could be to great help in all software development, modelling with abstract inheritance graphs and try to put the design as high up as possible in these graphs.

Figure 3. Using abstract devices



Now our code could be something like this:

```
DeviceFactory* factory = new EchelonDeviceFactory;
AbstractDevice* device = new Sensor(factory);
factory->CreateDevice(device);
...
while (...)
   factory->Initialize(device);
...
```

Observe how the hard-coded names now are moved from the main structure (the while-loop) to the creation of the objects. Again we use the is-a relationship to create an **`AbstractDevice`**, this time with a reference to the actual **`DeviceFactory`**. Instead of using a specification that is hard-coded against a certain device and manufacturer, you should write a specification that works for any device and manufacturer.

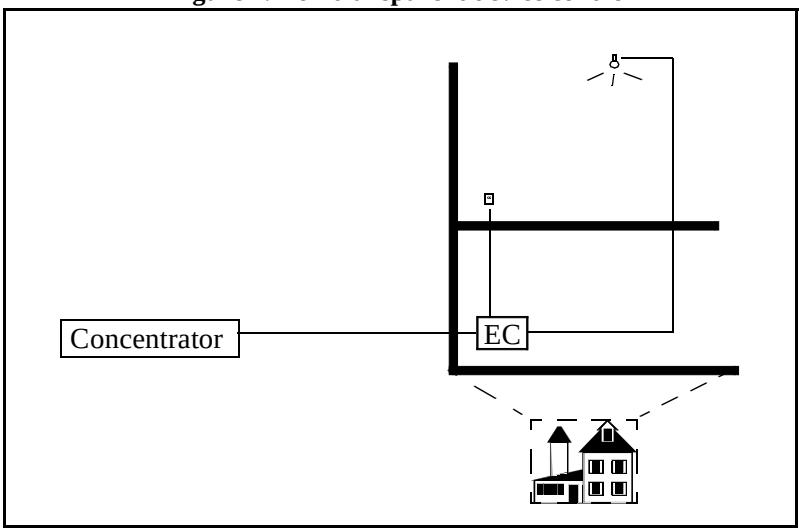
5 Using the proxy pattern

To fulfil requirement 2 and 3 we must hide all unnecessary details for the EC and build well defined and extensible interfaces. This is especially the case with the communication interfaces within the house and to the outside world. When communicating with devices in the house or the concentrator (figure 4) the EC should know nothing about the underlying communication details. Every physical device should instead be mapped into a virtual device object. You can see it as instead of working with the physical device you work with a corresponding virtual image. This abstraction creates more general structures with the domain specifics put as far out as possible in the module structure.

Several communication standards has been developed on these basis. The biggest one called MMS (Manufacturing Message Specification) was developed in the beginning of the eighties by General Motors to guarantee compatibility between computers. The major drawback with MMS is that the standard is quite complex and hard to implement on small computers (for example, the EC). DLMS (Distribution Line Message Specification) is a smaller specification of MMS with less functionality but better suitable for devices with limited resources.

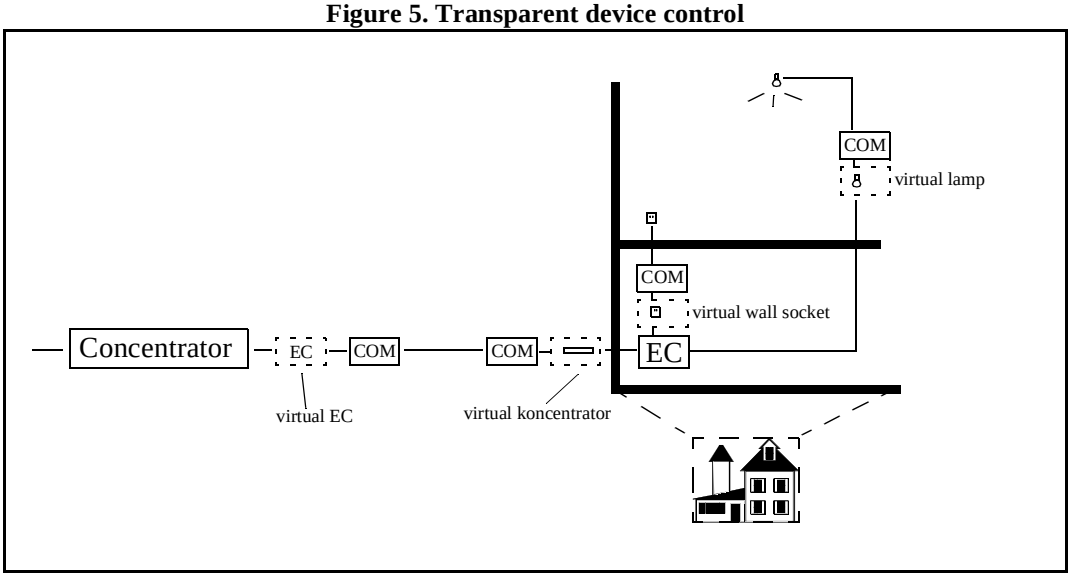
In figure 4 the EC has to know everything about the underlying details. When requesting services from the concentrator for example, the EC has to create a packet, initialize network connections, send the packet, wait for the answer and finally disconnect from the network. The previously scenario will be repeated for the communication with the wall socket and the lamp. The EC has to perform a lot of work, work that the EC shouldn’t care about.

Figure 4. Non transparent device control



In figure 5 we have created virtual device objects with references to the corresponding physical devices (the wall socket, the lamp and the concentrator). We also create a separate interface with the communication specific details, called COM. Now we can directly communicate with the much simpler virtual objects without have to care about the underlying communication details. For the EC the virtual devices seems to be local instances which directly can be reached through simple read and write methods. The main structure of the EC becomes smaller and less complicated using virtual objects than if using the physical objects directly.

Figure 5. Transparent device control

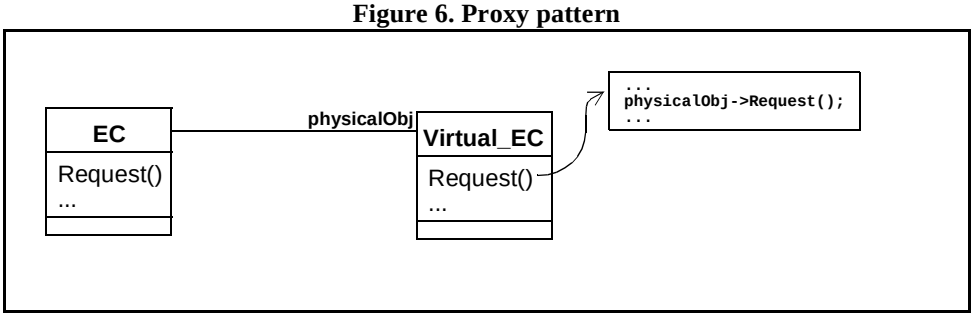


This transparent design should be used in this way to create as simple modules and components as possible. It should further be used in all communication links that are connected to the home system network. For example, the power supplier can control devices that is installed in the intelligent houses by directly call simple read and write methods defined in virtual EC objects.

Another important criteria using virtual objects is that the physical objects are called only when they are needed. For example, say that the power supplier is responsible for setting up different device configurations remotely in intelligent houses (turn off heating devices at 7 p.m etc.). In this cases we want to be able to easily set up and edit different device configurations without calling the physical device objects more than when its really needed.

What we have described in this chapter is in principle the **proxy** pattern (figure 6). This pattern uses a virtual image with a reference to a physical object which facilitates communication with the physical object only when its needed.

Figure 6. Proxy pattern



6 Using the reactor pattern

The home system network has a typical event-driven control flow where devices should react on input events sent out from other devices. It would be to great help if the EC reacted on events in the same way. It is further likely to believe that some of these events will happen at the same time. For example, two temperature meters sending data to the EC at exactly the same time.

The **reactor pattern** (figure 7) provide several major benefits for event-driven applications. It facilitates the development of flexible applications developed using reusable components. In particular, the pattern helps to decouple application-independent mechanisms from application-specific functionality. The application-specific functionality is performed by user-defined methods in the subclasses to the super class **`EventHandler`**. The **reactor pattern** facilitates application extensibility by separating the central event control mechanism from the application-specific functionality.

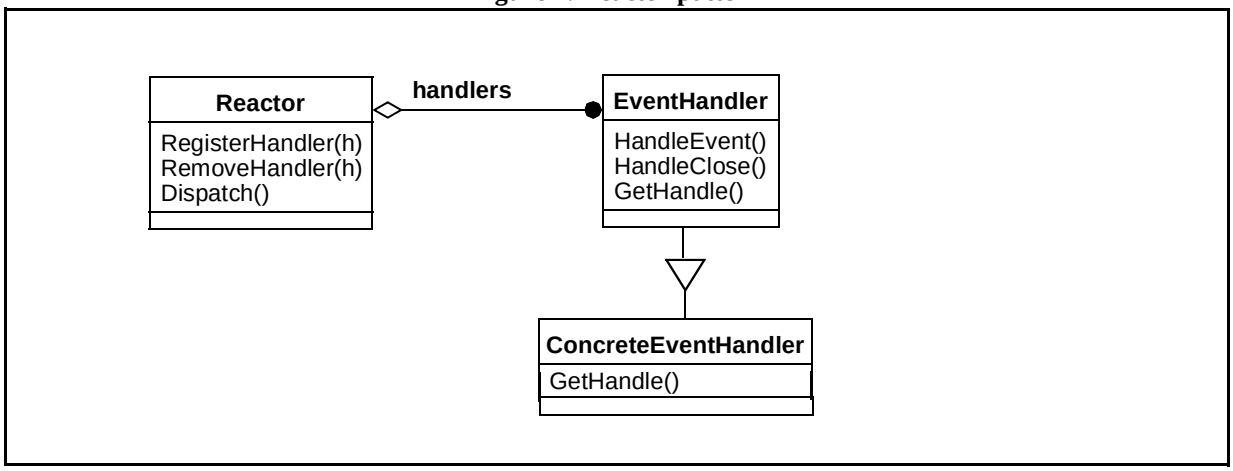
The reactor pattern is a server object that can control and distribute these events in a single-threaded way. Its important that the reactor never must block while reading, because there can be others devices who must connect and communicate within the home system network. Then you might ask your self why not create a thread for every unique client channel?

Using multi-threaded objects to implement event handling in the server object has several drawbacks:

- It may lead to poor performance on single processors.
- It may require complex concurrence control structures.
- Small operating systems normally don’t supports threads.

You should not use the reactor pattern if you want to send large amounts of data. Data packets with commands and meter values in home systems are although quite small.

Figure 7. Reactor pattern



The **`Reactor`** defines an interface for registering, removing and dispatching **`EventHandler`** objects. The class provides an application-independent mechanism dispatching many application-specific **`ConcreteEventHandler`** objects. To bind the **`Reactor`** together with these handles, the subclasses (**`ConcreteEventHandler`**) must override its **`GetHandle`** method.

7 Conclusion

Designing reusable software for home system network is not easy, this is because the abstract level of thinking and the many dependencies that exists in such systems. But by having an open minded attitude for existing design solutions and existing systems your chances for designing reusable products will be higher than if doing it all by your self. Just by knowing the intention of some existing design patterns can help you to quicker find reusable solutions.

8 References

[1] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, ‘Design Patterns: Elements of Reusable Object-Oriented Software’.

[2] D.C. Schmidt, ‘Reactor, An Object Behavioural Pattern for concurrent event demultiplexing and event handler dispatching’.

[3] Staffan Hägg and Fredrik Ygge, ‘Agent-Oriented Programming in Power Distribution Automation’.

[4] A.Kung, B.Jean-Bart, O. Marbach, S. Sauvage, ‘The European Home Systems Network’

Class Hierarchy Changes and Versioning

Erik van Campen
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: mpt95dvc@pt.hk-r.se

Abstract
Software has to be adjusted from time to time. Long-running systems require adjustable software components at run-time since the system may not be turned down. This can lead to a number of problems like the fixed behaviour at creation time for instances. The class hierarchy, the versioning of classes and the class types are also difficulties and will be part of an example which more problems contains. At the end of the example a set of data needed for a converter from one version of a class instance to another.

Keywords: Class Hierarchy, Long-Running System, Versioning

1 Introduction

Software needs to be adjusted form time to time for various reasons. For a long-running system this means that adjustments must be done while the program is running what is the case for for example insurance companies and banks which need to have their computer systems running 24 hours a day the year around. This situation demands more from the hard- and software than when the system may be down for a couple of times a year. A hardware demand is for instance a backup system which takes the control when the system fails or an aggregate which ensures power supply when the normal power provision drops out. A software demand is that the program isn't allowed to starve because of a bug in that program.

This paper addresses some of these problems for the object oriented paradigm which is very popular today. Here difficulties are for example the addition of new objects in the hierarchy and the fixed behaviour of an instance at creation time which is recognized in reference [02]: "In many object-oriented languages inheritance and method-combination have a fixed behaviour of an instance at creation time. This is undesirable in long-running systems, where run-time events can cause the behaviour of the objects in the system to change over time."

The paper continuous with chapter 2 "Arised Difficulties" which describes some problems which are also valid for long-running systems. The next chapter 3 "The Problem" contains the problem statement. Chapter 4 "Solutions" describes some general solutions found for the problems in chapter 2. The next chapter "Language Constraints" mentions some constraints which languages have that influence the reorganization of class-hierarchies. Chapter 6 "Reorganization" describes some kinds of inheritance, some points which must be considered when the version of an object has changed and two kinds of types. Chapter 7 "Issues in more detail" in which an example is used for the explanation of issues dealing with the problem statement. The paper ends with a small discussion.

2 Arised Difficulties

The world changes and so does the information in a computer system. For example when the information of a person who is a medical student has applied for a job after he has graduated then he probably will become a medical man. This transition of the role of the person should be made explicit in the information of that instance but this isn't the only reason why difficulties arise:

- the almost ever changing needs of users like new functionality in the existing software which can result in considerable changes in the software.
- experience shows that stable, reusable classes are not designed from scratch but the design is an evolutionary process. The management of the class modifications in a system where changes are made incremental and where the correct functioning of the system should be preserved two requirements should be met: all the changes in the object-oriented system must be properly propagated and no unnecessary changes should be made as a side effect.[03p326]
- the hierarchy constitutes a limited functionality because it can only be extended with subclasses. Many languages, for example Smalltalk, have a single parent object which itself has no parent. All in the system available objects are a direct or indirect child of this object, often called OBJECT.
- reusing software raises complex integration issues when the classes do not originate from a common, compatible hierarchy.[01p1,3] The programmer's knowledge of the contents and dependencies of a class must be accurate before it is reused but often the documentation and the program code aren't clear enough. Or names of classes must be changed in order to make it a part of the inheritance hierarchy.
  Here an important assumption must be made for the application of techniques like inheritance, genericity or delayed binding. Real-world concepts must be properly encapsulated as classes, so that they can be specialized or combined in a number of programs. Inadequate inheritance structures which misses abstractions in the hierarchy, too specialized components or a shortage in the object modelling may seriously influence the reusability of a class collection.

3 The Problem

Changes like the implementation of new versions or the addition of new components of software in long-running systems must be done while the system is running. This kind of changes in the system will probably lead to class reorganizations. The class reorganizations as a result of the dynamic behaviour of objects give difficulties in long-running systems.

The discussion in this paper is about three problems which are recognized in the papers [01,04,05] as issues which should be addressed when changes are made in object-oriented software components as starting-point. These related important problems with changes in the hierarchy are the class hierarchy; the version of an object; the type of the object. The second purpose is to define an abstract design of how versions can be handled is the software environment.

When, for instance, an object should be able to change from super class as a result of a message then a sudden change will make that the notion of the place in the hierarchy of some other objects is wrong and the program must still continue. Maybe the object has changed its type.

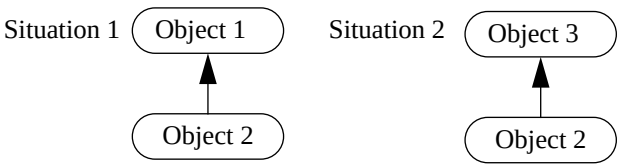


Figure 1: Object changes from parent.

In Figure 1 the object 2 is moved from parent object 1 to parent object 3. This behaviour influences also the child objects of object 2. When the move as a result of a message is done then some objects could still need the information and behaviour of the older object hierarchy. The hierarchy can in this case contain knowledge of both objects. The result of an account calculation must be evaluated two times. When the third calculation starts already 2 calculations are stored in 2 instances of an object: the first version and the second version.

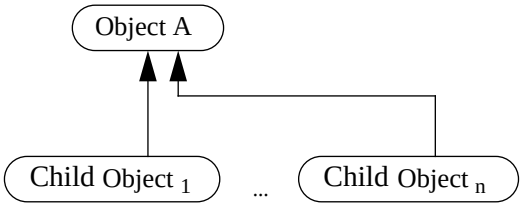


Figure 2: Object 2 has n children.

An implementation may be the introduction of a default object [01p20] which takes care of the dispatch to the child $object_i$ of object A in figure 2. This is a form of reflection (meta object) where object A represents a group of versions of the same object. Before a child object is accessed object A is called and according to the information from the sender object A will call a child object with the right behaviour.

An example for the identification is the use of environment enquiry for the handling of active change. The behaviour of the system changes according to what it finds in its environment, [03p330] like wordprocessors which use an available font, or an opearting systems who adjust their notion of available devices.
This is done when the software is started and is also not useful in long-running systems. Long-running systems would have to inquire the available environment continuously.

A suggestion is the use of change absorbers like RPC mechanisms, database interfaces such as ODBC (Microsoft) The aborbers only transmit the important changes across boundaries and suppress the propagation of less important changes. Another example is the object transmission mechanism of CORBA. The intermediate machinery performs transformations to recover from representational changes across the interface. [03p329]

4 Solutions to Arised Difficulties

Chapter 2 “Arised Difficulties” contains some common problems which often also hold for long-running systems. In this chapter solutions to these problems are categorized but class changes in this kind of systems have more difficulties. Here are a number of proposed solutions for the control of the environment in databases which can be divided in:

- Tailoring which is changing a class slightly definition in order to make subclassing possible. The subclass allows the programmer to replace unwanted charateristics.[01p5] The functionality of a hierarchy can be adjusted by derivation of new classes without actually changing existing objects. [05p115] For example the replacement of a method by a method with a more efficient algorithm.
- Surgery primitive. A change in the modelling of an application domain results in adjustments of the classes which represents the real-world concepts (often implemented as abstract classes). This kind of operations change the class hierarchy. The consequences of such a change must be easily determinable so that the consistency of the class hierarchy is maintainable.

- Versioning keeps track of the history of class modifications. Normally the versions are stored in databases which enables the programmers to keep track of the creation and the location of the software components. Some important issues with versioning are: structuring the different kinds of versions, like mutable/immutable, public/private; determining which version is used when an object of a particular class is instantiated; deciding what are the operations that justify the creation of new versions. [05p116]
- Reorganization of a class library is needed when the library is significantly changed. Large reorganizations will cost a lot of process time which is for long-running systems not desirable. Small changes to hierarchy will lead to small process time.

An other problem in this area is related to the evolution of classes. After an update of an instance of a class the representation of the instance is modified which can be addressed by

- Change avoidance which means that the system prevents class changes of existing instances as a result of modifying a class.[01p3,4] The notice of change in systems should be limited to the new features according to many users. [03p328]
- Conversion transforms objects which are affected by the class change to conform the new definition of the class. [01p3,4]
- Filters hide the differences between objects belonging to several variants of the same class. The instances are encapsulated in objects that extend their normal properties.[01p4]

5 Language Constraints

Reorganizations are limited by a number of integrity constraints often called the class invariants which differ between languages. The paper will not make a comparison between languages. The examples in the paper are made as language independent as possible.

- Representation invariant. The properties of an object, like the attributes, must reflect those defined by its classes.
- Inheritance graph invariant. The structure of the inheritance dependencies is often restricted to a connected, directed graph without circuits. This kind of structure has a root object.
- Distinct name invariant. The classes, methods and variables must be distinguished by unique names. A variation on this constraint is that the children of a class are uniquely identifiable other links may have the same name.
- Full inheritance invariant. A class inherits all attributes from its ancestors, except those that are redefined. A precedence scheme which selects one attribute as the inherited solves the naming conflicts because of multiple inheritance.
- Distinct origin invariant. An attribute which can be inherited via different paths is only inherited once.
- Type compatibility invariant. A type redefinition of a variable in a subclass must be within the type domain of its superclass. The pre- and post-conditions of a method may be redefined, as long as the new pre-condition (new post-condition) is weaker (stronger) than the original one.[01p6]
- Type variable invariant. The type of an instance variable must correspond to a class in the hierarchy.
- Reference consistency invariant. The object which is referenced may for the time it is referenced not be modified.[01p11,12] This kind of rules can be found in every database and there it is called a commit-phase.

When the distinct name invariant is limited to the children of a class then a child class can only be changed with the danger to create a name conflict after the reorganization.

6 Reorganization

The hierarchy should reflect the application domain [01p23] The proposed automated reorganization algorithms perform only strict structural transformations, for example the algorithm in reference [04] or [05], so their results must be inspected after a reorganization because of the lack of knowledge about the application domain. [05p130]

As stated earlier the related important problems with changes in object-oriented software components are the notion of the class hierarchy; the versions of an object; the object type. Each of these important problems will be described in the next sub chapters.

6.1 Hierarchy

The hierarchy assumed here is the class inheritance hierarchy. An other approach is the prototyped based language. The class hierarchy has the form of a graph but many languages demand that the graph complies to the inheritance graph invariant. A further decomposition of languages is the division in languages with single (Modula 3) or multiple inheritance (C++, Eifel, Oz).

With single inheritance the class inherits from one parent. This class differs from the parent because it is

- an extension. The class adds instance variables and methods to the parent. It is also a refining of the parent.
- a variation. The class redefines some features of the parent.
- a specialization which is a combination of an extension and a variation.[02p28]

A class in a language with multiple inheritance can inherit from more than one superclass. Two operations could in this case have the same name because of inheritance along different paths which can lead to a conflict. For a language this means that the language has to choose the order of accessing the superclasses. These approaches can be categorized in

- The language calculates a class precedence list by depth-first traversal of the inheritance subgraph rooted at the class. The method with the highest priority will be chosen. A class precedes its superclass in the precedence list and the local ordening of any class's precedence list is preserved. This local precedence is determined by the specified order of the classes immediate superclasses as listed in the class definition.
- The programmer must code in the metaclass of a class the algorithm for computing the class precedence list from the local precedence of the class being defined. When no algorithm is specified then the first mentioned algorithm is used.
- No assumption of the precedence of the superclasses is made. The programmer must explicitly state from which superclass it wants to inherit.

These in languages implemented approaches can be problematic when for example a method name in a class changes. In the second approach the programmer may have to change its metaclass when the method names are explicitly stated. Or he has to rewrite some classes because they don't recognize that method any more although they depend on it.

The reorganization algorithm [04] based on the demeter data model differs from the approach taken by Casais [05]. The demeter approach tries to reduce unwanted coupling between classes by suppressing certain kind of attribute dependencies. The reorganization is based on a collection of classes which are recasted at a time.

The approach Casais took is an incremental reorganization so small parts of a collection can be recasted. An incremental reorganization is more suitable for long-running systems.

6.2 Access to the right version

Versioning is interesting for managing class development and evolution. Recording the history of classes enables the programmer to try different paths while modelling complex applications. This can also help track on the of various implementations of the same component for software environments and hardware platforms. In long-running systems this could be useful for an insurance company which uses different calculations for the insurance benefit based on the same accident but on different policy numbers.

The prevention of incompatibilities between parts of the hierarchy belonging to different versions is part of the maintenance of the software. A complication of a new version of a class is that the installation of a new variant of a class can have influence on the tree of sub-classes. A thorough analysis of the version change gave a couple of directions for this problem:

- The change of the interface of a class will result in the a new version of all the classes depending on this interface.
- When internal parts of the class are changed, like the methods only visible to subclasses or the type of its variables, then the version change can be limited to its subclasses.
- An adoption of the realization of a method won't lead to new versions for other classes. The internal change of the method doesn't affect other definitions.[01p21]

6.3 Object Type

The type of a variable, a method or a class can be static or dynamic. For a static typed language the type of the variable, method or class is known at compile time and for a dynamic typed language the type of a variable, method or class isn't known until run-time. Often both mechanisms are used.

Many languages that perform static type checking allow a variable of type S to be assigned to a variable of type T if and only if S is a subclass of T, thus equating inheritance with subtyping.[02]

7 Issues in more detail

The previously described problems and solutions will be addressed in more detail in this chapter using examples. Instead of every time mentioning the term class-object the term object is used.

Assume each object in a long-running system has a filter object, called Controller, which filters the input-messages for the object at run-time. The system has to know the new version of an object. The programmer must add the version number to the new object before the controller gets the notion of the object. The controller dispatches the message to the object that matches the version number. In the figure beneath is also an object which gives the message to this kind of control object.

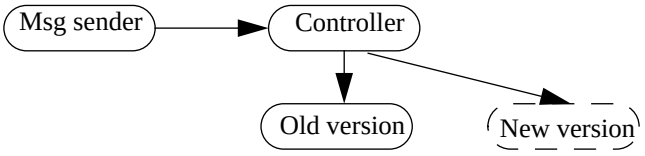


Figure 3: Message sent to the version controller.

(The arrows follow the path of the call.)

The version control object must know for which version of the object the message is. The object must be identified. This identification can be done in two steps. First the control object can be called with only the

name of the destiny object assuming that each name can be identified uniquely. Second the object with the right version can be recognized by the tuple (destination identity of the message, destination version). The return to the Msg sender demands that also the identity of the Msg sender is sent so the actual message must at least contain the quadruple (destination identity of the message, destination version, sender identity of the message, sender version).

7.1 Compatibility

This way of addressing the version problem is very simple since an object could change from name even though all the properties of the object stay the same. When the name is used as identifier then messages for the old object with the old name can't be addressed any more unless the controller understands the new name as a new version of the object. It is better to give the object-controller, also the object, an unique identifier during compile time which makes renaming of an object during run-time independent from the identification. But this scheme is not enough for addressing the degree of compatibility between the instances of the old and the new version. New versions could have properties added or deleted. In this there are two kinds of compatibilities the backwards and the forwards compatibility. A version of a class is backwards compatible with an older version when this class can use the instances of this older class. A version of a class is forwards compatible with a later version of the class when the class can use the instances of the later version of this class. The message sender can also have some versions. The version of the message sender can be compatible with only the latest version of the message receiver or with an older version of the message receiver.

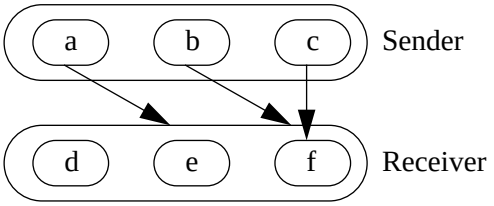


Figure 4: A number of versions of the sender can call a number of versions of the receiver

In figure 4 a, b and c are different versions of the sender and d, e and f are different versions of the receiver. The version a is compatible with e and the versions b and c with f. The receiver's version d is not consulted by the sender.

An important feature to mention here is polymorphism. The feature allows for flexible software elements which are manageable to extension. A polymorphic operation has a type parameter which determines the to invoke object. An invoked operation dependent on one parameter is called simple polymorphism and multiple polymorphism is the ability to execute different operations based on the types of more than one parameter. The receiver may decide how to respond to the message. [02p32]

The introduction of this concept must be done carefully because of the complexity. The complexity can raise dramatically when the developer wants to change just one non-leaf object. As a result the whole subtree may also have to be adjusted.

7.2 Conversion

The storage of the versions in the running system is not always necessary. A natural approach to prevent incompatibilities of instances is to transform them to the version in use. During this process no information may get lost and especially for long-running systems the normal operation may not be endangered. Sometimes instances may not be converted because of legal reasons or conversion is not necessary. The modified class makes the use of the older instances obsolete. Conversion of all the to change instances can be done

directly after the installation of the new version. The objects in the object store must all be loaded, converted and stored which cost a considerable amount of time. This method is not applicable in long-running systems. A secondary system could make the conversion where after the new version will be introduced, but the operation must be interrupted while the object in operation could belong to the changed set.

7.2.1 Lazy Conversion

Or the system can be supplied with a lazy conversion algorithm. An instance is adapted the first time after the introduction of the new version, but an object stored in an object store may not be accessed for a very long time. The access of the before mentioned object will lead to an amount of conversions where after the object is accessed. The object will then be inactive for an unknown time period. Actually a version-conversion-routine must be in the system until all the instances for that object are converted to a later version.

When the conversion routine don't have to exist after the conversion of all instances then the total number of available instances for each version should be stored in the system. A conversion decreases this number for that particular version so that when the number becomes less than zero, when the count start with zero, the version could be removed until then the controller should contain the version.

7.2.2 The Converter

The long-running system needs some kind of lazy conversion because this kind of conversion doesn't ask for a much processing time in a very small period of time. The normal processing may not be endangered by this kind of version changes. The earlier mentioned controller had knowledge of the different versions which made the program's class inheritance structure very untransparent. The new version should not have this problem.

In the figure beneath the sender sends a message to the controller of the object (arrow 1). The object calls the instance which knows its version. When the instance is incompatible then the message holds and the conversion controller will be called (arrow 2). The conversion controller gets the identifier and the version of the object instance and searches the right converter (arrow 3). Each object has also an own conversion controller. The converters only transforms an instance to a later version of that object instance. Instances of other objects will also be changed when needed. The conversion of the object may take several conversions after each other which are in the figure Converter_1 till Converter_n (arrow 4). Here after the object instance is executed.

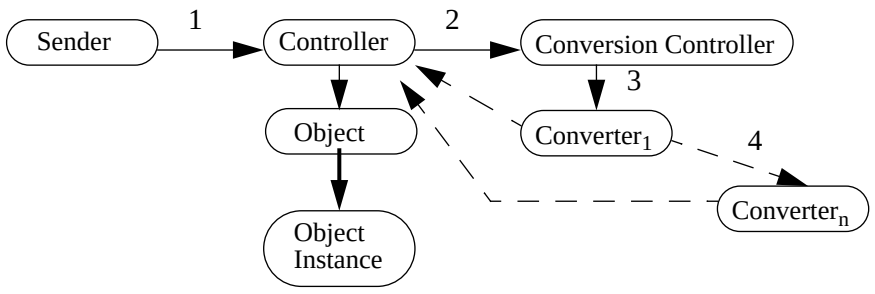


Figure 5: Control of conversion of instances of an object.

The converter needs information to carry out its task. The information needed when properties are changed during conversion is for example

- the type of the class, the type of the methods of that class and the type of the internal variables.
- the new implementation of methods or instance variables.
- the notion of the hierarchy in order to address changes which influence other objects.

- a mechanism not to loose properties which are not valid for the current version of the instance.
- the identifier of the instance and the version of the instance.

8 Discussion

The problems mentioned with class changes are just a subset of all the possible problems. In this case of long-running systems will have to use incremental change techniques, like lazy conversion of instances or incremental hierarchy adjustment algorithms. The normal processing may not be endangered.

Conversions of instances from an older version to a later version has to done by converters. Instances which aren't used for a long time may need several conversions before the version is the same as the latest. This means that several convertor versions have to be in the system at the same time. The use of reflection can keep track of the versions and of their converters which transform the instances from one version to another per object.

Some articles talk about the role of an object. The role structure differs from the class inheritance structure in that it addresses how a group of objects work together to solve a problem. As far as we know no reorganization algorithm for this kind of structures are made. We neither know what the influence of class changes on this kind of hierarchies are.

The paper contains no assumptions about time constraints. The reorganization of the inheritance hierarchy or the lazy conversion to a later version can still take too much time.

A well known reorganization algorithm is "The law of demeter" which can be automated [01], but the "class form" of the law can't be fully effective at compile time for untyped language: since objects manipulated by the language expressions are not to belong to a certain class. Nor are the changes made by this algorithm always natural.

9 Acknowledgements

With thanks to Magnus Egnerfors for the discussion in the middle of the night.

10 Reference List

[01] Eduardo Casais (1990). Managing Class Evolution in Object-Oriented Systems.
(c) Dennis Tsichritzis et auteurs. Tous droits reserves.

[02] Josephine Micallef (April/May 1988). "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages", Journal in Object Oriented Programming page 12-34.

[03] M.P.Atkinson, D.I.K.Sjøberg, R.Morisson (1993). "Managing Change in Persistant Object Systems", Lecture Notes in Computer Science 742. page 315-338

[04] Paul L. Bergstein and Walter L. Huersch (1993), "Maintaining Behavioral Consistancy during Schema Evolution", Lecture Notes in Computer Science 742, page 176-193

[05] Eduardo Casais (1992), "An Incremental Class Reorganization Approach", ECOOP 1992, page 114-132

Implementing dynamic behaviour in object-oriented languages

Björn Nilsson
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: mpt95bni@pt.hk-r.se
www: http://www.pt.hk-r.se/~mpt95bni

Abstract
When developing software, it is natural to think of a system with different states and state transitions. This dynamic behaviour is modelled along with a model of abstract data and perhaps a model of the data flow. Although these three different models are used to describe a system only the abstract data (object model) has language and compiler support. There is an obvious danger that features described in the dynamic model only, disappear in the implementation. This paper presents language support of dynamic behaviour such as states and transitions. The main usage is C++ in combination with the Object Modelling Technique (OMT).

Keywords: states, dynamic behaviour, C++ language extension

1 Introduction

An object oriented system is a collection of objects in an environment. These objects interact by sending messages to each other. These statements tell us that the system consists of data structures that can be related to each other at different moments. To reveal this, the system has been analysed and designed and there after been implemented and executed.

First the classes, their relations and methods were defined and put into an object model. To describe how the system behaves in run-time, a dynamic model is used. This model identifies the states a system could enter and how a transition from one state to another is done. Each state describes what should happen when it is entered, while it is valid and when it ends. Sometimes an functional model is used, this model describes the data flow in detail.

At the beginning of the implementation, the software engineer consult the analysis and design documents and implements the system - well, the object model anyway. The reason is that only the object model has language support, i.e. classes, methods and attributes.

This is a danger! When the system development is moved from analysis and design into implementation some dynamic behaviour could be lost due to lack of mapping of the dynamic model in the programming language. Because time related (i.e. dynamic) constructs can’t be implemented straight off it is common that these aspects are badly implemented and won’t work properly as stated in the design.

System errors due to this information loss could be: methods that are invoked too early or too late, or in illegal states such as an get-method in a list class that is empty and many other occurrences when time, execution order and states are involved.

As long as the programming languages lack of dynamic behaviour constructions, these errors can occur and confuse both testers and programmers.

This is the outline of the paper: chapter two defines the word state and explains two different interpretations. Chapter three consider what exists in practice on the state issue. Chapter four discusses the core problem and the object oriented paradigm. Chapter five discusses different issues that arises when using states. Chapter six presents ObjectPhases, a way to implement states in C++. Chapter seven is a conclusion followed references.

2 Definition of state

In the object oriented world there are two interpretations of a state. Firstly and most common, a state is all the values of the attributes of an object at a particular moment. Secondly and used in this paper: a state is a limited time period when certain actions and activities takes place.

2.1 The state of a system

When a state is used at system level you have a total view of the complete system. Depending on abstraction level, different amount of states could be defined. You could for example define two major states (e.g. System_Idle and System_Running). But there could also be a large amount of small states (e.g. Waiting_For_Key and Calculate_Sum) and similar.

This approach suits the dynamic model exactly because dynamic models captures the whole system.

2.2 The state of an object

One approach is ObjChart [Gangopadhyay et al.] it uses finite state machines in the object methods in order to execute it. The FSM's are used to capture how the object reacts when it receives messages. Methods are used as "short-hand" for FSM's.

3 In practice

How is states used in practice? There are many different kinds of examples, both in analysis and design and in implementation. This chapter will present some existing work.

3.1 Analysis and design techniques

Most design techniques uses a dynamic model to model states and events.

3.1.1 OMT

OMT [Rumbaugh et al. 1991] is a widely known modelling technique. Some of its parts have been discussed earlier. The dynamic model consists of a number of state diagrams, one for each class with important dynamic behaviour. Scenarios are also used. A scenario is a sequence of events. This sequence is abstract and printed in english sentences. The event trace diagram shows a scenario but includes the objects. In this diagram you could actually see which object are doing what and in which order. OMT also supports concurrency and synchronization. States and events are related to classes and methods, respectively.

3.1.2 Object-oriented analysis and design

This technique developed by Grady Booch [Booch 1994] presents dynamic semantics that consists of two diagrams: the state transition diagram and the interaction diagram. The state transition diagram is similar to

the dynamic model in OMT. The interaction diagram is used to trace the execution of a scenario, how instances of different classes sends messages to each other. This diagram is a generalization of the event trace diagram in OMT.

Both OMT and Booch technique uses the same terminology and have very much in common.

3.1.3 Statecharts and Finite state machines

Statecharts and finite state machines are the concepts used in most analysis and design models. These are intuitive and powerful but are not sufficiently powerful as an object-oriented modelling tool nor are they well integrated in the object-oriented model [Bosch 1994].

ObjChart as mentioned before are using FSM's. An ObjChart model consists of a part-of hierarchy of objects with behavioural and functional relations. The behaviour of these objects are controlled by FSMs.

Statecharts was originally defined by Harel [Harel 1987].

3.2 Programming languages

Few languages support states. Those who are, have certain drawbacks that don't solve our problem with implementing a dynamic model. Only LayOM is briefly presented, other languages does exist but are not included here.

3.2.1 LayOM

LayOM [Bosch 1995] consists of five major components: variables, methods, states, categories and layers. States includes concrete, abstract and active states. LayOM stands for Layered Object Model and the concept of layers are heavily used. The layers model relations between objects and can restrict access to them depending on the current state.

4 The problem

The core problem is that states are not properly implemented and that is due to lack of language and compiler support.

4.1 States in design phase

Many design methods include some sort of dynamic model. This implies that this concept is necessary. Every model adds information to describe the system. There fore it is important to follow the models that are designed when the system is implemented.

4.2 States in implementation

When a system is implemented its classes and relations are specified. These specifications are static and lack proper real time behaviour. Real time and state behaviour could be implemented with these static components but the result may not work properly if at all.

The dynamic behaviour have to be implemented without any support from the dynamic model - then what's the point of spending time to make the model?

4.3 Solution

The solution is to provide language and compiler support for states. The expressions used in the dynamic model must be mapped into the source code.

4.4 States and object orientation

“A state is a limited time period when certain actions and activities takes place”

Is this definition and its concept really object oriented? Isn’t it a function oriented thinking? Compare the dynamic model and a real function oriented model such as a data flow diagram, what the difference? A data flow diagram and similar lack of an object concept and focuses on (less abstract) data flows. A dynamic model is a description of objects in motion. An object with dynamic behaviour is nothing illegal.

States could be implemented with object migration. When an object transits to a new state, the object is migrated to a new object. Every object has data members that fits the actual state. This idea is perhaps more *object*-oriented but has its drawbacks. For example, the system becomes very big. The class hierarchies will grow. Say a simple system with five objects and four states, this will result in 20 different objects and most likely another five super class objects. See Figure 1. An object for each state.

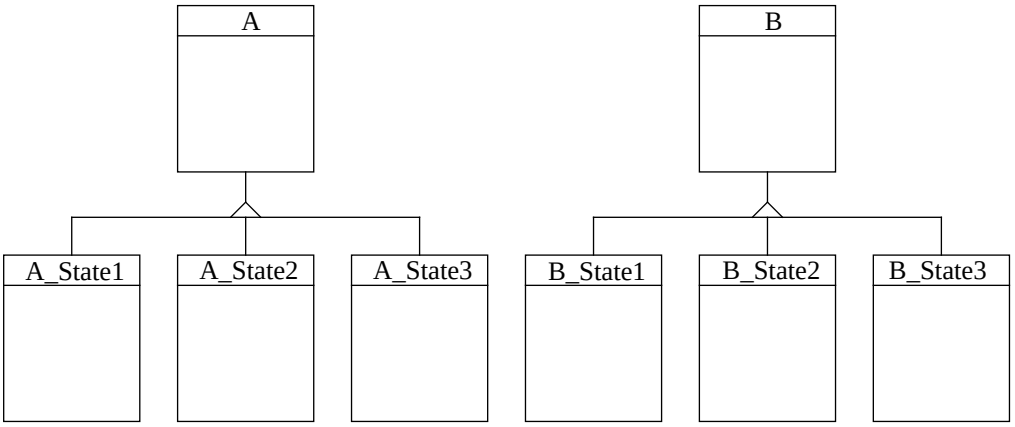


Figure 1. An object for each state.

A better approach is to see the possibility that the one and same object can enter different states. It is the same object that changes state. This would instead look like Figure 2. Same object for all states.

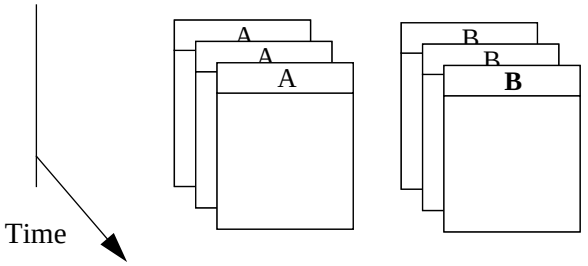


Figure 2. Same object for all states

5 States

The following chapter discusses some important issues that are problematic in the implementation aspect of dynamic behaviour.

5.1 State applicability

Are usage of state more or less applicable for different applications? Applications that are pure functional with no communication and no time constraints what so ever wont need a state concept. These applications are fortunately quite unusual.

Applications that have a dynamic behaviour such as real time systems are of course best suited.

But generally one could say that every system that is designed with a dynamic model that actually adds information to a system ought to be implemented using states.

5.2 Placement of states

If states should be used, where should they be placed in the system? One approach is to model states as objects, see [Sane et al.] or as [Gangopadhyay et al.]. This means that the state concept is only used in the smallest parts of a system, the objects.

The approach used here is states on a global level. States at global level means that the state concept is on system level. All states are known in the complete system.

States and transitions to other states should be visible everywhere in the system. A system could enter a new state somewhere in the system and all objects notices when this happen. Depending on the specification of this new state in the class definition, certain actions and activities could be performed and a new set of rules or constraints are valid.

5.3 Abstraction level

A state is a part of an abstraction. What guidelines should be followed? A system with a very large amount of states becomes hard to manage and overview but a system with very few may fail to provide anything useful to the system. The solution is that it is a decision of the developer. The same abstraction level used in analysis and design should also be implemented else there would be risk of information loss.

OMT supports substates and superstates. These are seen as generalization of states and an object can only belong to *one* state at the same time, the substate or the superstate. See Figure 3. Superstates and substates for a transmission dynamic model for an example of this. Here, *Forward* is a substate to *Transmission*.

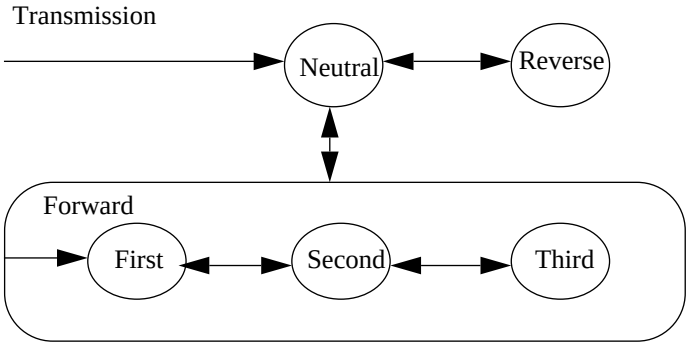


Figure 3. Superstates and substates for a transmission dynamic model

5.4 Behaviour constraints using states

A state can be seen as a set of constraints. In every state there exists some restrictions of what could be done, what methods that are legal to use and those who are not. Attribute restrictions is another way to constrain a behaviour. Attributes (e.g. guards in OMT) could be used to decide whether a transition to another state should be done or not. This is done by testing if a condition is true or false and the system will only transit if this condition is fulfilled.

5.5 Meta-behaviour

With a general state definition available, other behaviour such as reflection and meta-behaviour could be implemented. Such constructs can be obvious and naturally incorporated on top of a state.

Along with a state declaration and or guards as mentioned before, meta constructions could be added.

5.6 Supported dynamic model features

The OMT model has some features that is used to describe a dynamic model. What features should be supported?

The state is of course one, events that transits to another state is also necessary. OMT also provides activities, that is the task that represents the state. Actions are instant operations, commonly executed at the beginning or end of a state. Guards are boolean conditions on the transition events. In order to transit to another state the guard criteria must be fulfilled. Event attributes are attributes that act as input to the next state. OMT also supports substates, superstates and nested states.

5.7 States as behaviour verification

One of the major benefits of implemented states would be a verification that the system behaves as defined in the dynamic model. While compiling a program, illegal behaviour could be discovered and corrective actions could be made.

For example say that an illegal method is called in a certain state, the compiler find this error and complains. The code is state-checked to see if it is correct according to the dynamic model.

5.8 State transitions

States could change any time depending on the defined transitions. There are two ways of enter a new state, an explicit call or an implicit call.

The implicit call is the most appropriate and is made when a method is executed somewhere that lead to a transition according to the state definition. Every method that is invoked must be checked to see if this call would lead to a state transition. There fore it is necessary to define before compilation what methods that are legal and what new state they may enter. This approach would also provide possibilities for the computer to analyse a state-transition graph. This graph could be used to examine the structure of the dynamic model and be used in run-time to handle the state flow.

An explicit call must be implemented by the programmer and there is a risk that it would be forgotten or misplaced. This explicit call don’t need any form of checking whether a special method invocation leads to a state transition as mentioned before, but is a bad solution.

5.9 State definitions and source code

In what way should ordinary source code and variables cooperate with state definitions? A state definition could be seen as a strict definition of behaviour or a sort of function with executable source code.

The first one, the behaviour-only alternative, don't include any source code but only the general structure of states. This includes all states and their transitions, what methods that leads to a state transition and to what new state. There is no data sharing between the object and the state.

The second one, the behaviour-and-source-code alternative, include the behaviour (state and transition definition) and also source code such as variables and test-cases such as if-then constructions. The source code is used to complement the state.

Guards, as mentioned before, are boolean expression that must be fulfilled by the system in order to change state. This must be implemented with some form of source code, the boolean expression could involve a member variable or similar and must access the object. Variables that are local to a state is possible but not practical. Variables used in state definitions should be class members.

As an examples these can also be used in cases when a state transits to another after an amount of time. Say, that a system is in a state called Busy and after three failed tries to invoke a method (let say that the guard expression couldn't be fulfilled) it moves to a new state called Error.

Issues like entry and exit actions belong to the behaviour-and-source-code alternative. Here the state concept have to execute code. Entry and exit actions are methods invoked when a state is entered or leaved. This implies that the state concept are taking over the thread of control for a while. Similar to this is the action on event principle. This means that a certain action (method) is invoked while an event (state transition) is performed. If that is good or bad is left for discussion. The good thing is that it maps to the dynamic model and the bad thing is that the program flow is a bit harder to follow.

5.10 Inherited states

If states are defined inside classes as methods and variables are, these are inherited just like everything else. All states defined in a superclass are derived and used. The states can of course be overridden by the subclass and instead these are used.

6 ObjectPhases

To extend C++ with state behaviour a new grammar must be added. This extension is called ObjectPhases and exists only in theory.

A state is declared inside each class and defined the same way as the class methods (i.e. in a header file). The state declaration is seen as a fourth access method (after private, public and protected). After the state keyword and a colon all available states are listed and separated by a semi-colon. These states are the only one supported by this class.

```
class C
{
      private:
         int v;
      public:
         C();
         ~C();
         void WaitForKey();
         void Operate();
      state:// State definition
```

```
         Idle;// Supports the Idle state
         Busy; // Supports the Busy state
         End;// Supports the End state
};
```

Above, a class C is declared and inside it there are a list of three supported states. All states are ‘global’ for all classes, that is every class should be able to support an existing state (if it is applicable). All states are also named differently to separate them. Two states named equally would generate an error just as two equally named classes.

Below, there are three definitions of states. Note the similarities of a state definition and a method definition. Each state definition consists of two parts: the legal and the event part.

The *legal* part declares a list of legal methods (usually class methods). Only these methods are allowed to be executed when the system is in this state.

The second part is the *event* part. This part declares legal transitions, to what state a method (from the legal part) transits. Only these methods could generate a state transition (in this class, of course other classes could have other or same event transition declarations). Of course the state must be known to the class, or else an error would be produced.

```
C::Idle    // Declaration of class C’s Idle state
{
      legal:// List of legal methods when Idle
         C();     // The constructor is legal
         void WaitForKey();// And this method also

      event:// List of possible state transitions
         WaitForKey() -> Busy;// This method implies Busy
}

C::Busy    // Declaration of class C’s Busy state
{
      legal:
         void Operate();
      event:
         Quit() -> End;// Quit method leads to End state
         ~C() -> End;// Destructor also leads to End
}

C::End     // Declaration of class C’s End state
{
      legal:
         ~C(); // Only the constructor is legal
      event:// No possible state transitions
}
```

Guards can be implemented in the state declaration in the event part. Every listed event is proceeded with an if-then expression. Every time a method is invoked that leads to another state, it must pass this if-then expression (the guard). If it succeeds the new state is entered else it is not.

```
C::Busy    // New declaration of class C’s Busy state including guards.
{
      legal:
         void Operate();
      event:
         if( v == 5)      // Guard expression: if the member variable v equals 5...
            Quit() -> End;// Quit method leads to End state only if v eqauls 5
         ~C() -> End;// Destructor also leads to End
}
```

7 Conclusion

C++ and many other object oriented languages lacks support of dynamic behaviour. This is one reason to programs that executes abnormally. Even when a dynamic model exists that represent the correct dynamic behaviour of a system it is hard to actually implement it. There fore it is necessary to extend the language with dynamic support such as states. The best approach is to define constructs that can directly map the dynamic model into the source code. ObjectPhases is an extension that easily supports states, state transitions and guards. Entry and exit actions could easily be added, but the solution for these could be argued because it will disturb the program flow.

While introducing a new solution, there will arise new problems. The disturbance in program flow for entry and exit actions is one. Another, and related, problem is how much source code that should be used in states. The alternatives behaviour-only and behaviour-and-source-code is an important issue. This state extension requires some form of over head to solve to state flow but is defined during compilation.

Although this haven't been tested in reality, the presented solution is very simple and easy to use and will provide C++ programs with states. This lead to better dyanmic behaviour and less errors.

References

[Booch 1994] G. Booch, 'Object-Oriented Analysis and Design with applications'

[Bosch 1994] J. Bosch, 'Abstracting Object State', Department of Computer Science and Business Administration.

[Bosch 1995] J. Bosch, 'Layered Object Model - Investigating Paradigm Extensibility', PhD. dissertation, Department of Computer Science, Lund University, November 1995.

[Gangopadhyay et al.] D. Gangopadhyay and S.Mitra, 'ObjChart: Tangible Specification of Reactive Object Behavior'

[Harel 1987] D. Harel, 'Statecharts: A Visual Formalism for complex systems', Science of Computer programming 8

[Rumbaugh et al. 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, 'Object-oriented modelling and design'

[Sane et al.] A. Sane and R.Campbell, 'Object-Oriented State Machines: Subclassing, Composition, Delegation and genericity'

A description of a Dynamic and Interactive Language

Magnus Egnerfors
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: mpt95meg@pt.hk-r.se
www: http://www.pt.hk-r.se/~mpt95meg/

Abstract
The Dynamic and Interactive Language (DIL) is a language that supports the programmer with the ability to change the behaviour and structure of the program during run-time, either by having programmed the changes within the program or interactively accessing the program structure through the programs serviceport. This type of interactive changes of the program are compiled at run-time so that the system isn't needed to be brought down when the changes are about to be incorporated. DIL is based upon the language SELF.

Keywords

DIL, Interactive language, dynamic language, run-time compilation

1 Introduction

When writing a program today after having gone through the analysis and design it becomes time to start implementing the program in a language. This is often done by first writing the code and then compile it to create an executable program (e.g. C, C++). An alternative way of doing this that is rather commonly used is the use of an interpreter of the program (e.g. Smalltalk).

Another solution is that of dynamic linking/unlinking of modules during run-time that allows to change a part of the system without the need to recompile the whole system. This approach is taken in the persistent C++ language E [Vemulapati 95].

Still none of theses languages allows the programmer to really change the behaviour and structure of a program during run-time. This paper intends to show a possible approach for a language that allows the programmer to change the behaviour and structure of the program when it actually is running. This will give the programmers better control over what is happening within their programs and they don't need to have their work interrupted because the need to compile the code. This is actually achieved in SELF [Bosch 94], [Ungar & Smith 91], but it stills misses some of the points that we find important.

Within this paper there will first be a description of the problem (chapter 2 Problem) and some of the cases where this kind of system might come in handy. After that, there will be a general solution (chapter 3 Solution) that briefly explains the steps taken to build a language that works. In chapter 4 SELF short introduction a short introduction to SELF will be given. In chapter 5 The Dynamic and Interactive Language (DIL) a more detailed solution for the problems will be given.

Within this paper the notion of interactive is in the way that the programmer has the ability to change running program at any time he wishes, and that the results will be visible within moments of time.

The definition of dynamic within this paper is that the programmer has the ability to change the programs behaviour and structure while it is running.

2 Problem

When/where is a dynamic language, that allows interactive changes to be made, needed?

One of the examples that first pops into the mind are for those systems that has to be on-line 24 hours a day all the year around (e.g. bank systems where the control of the cashflow is crucial, nuclear power plants can't allow that the programs are needed to be taken down because a minor change has to be made, and shutting down the plant is a to cumbersome procedure to do, for just adding a fix or a new necessary feature). These changes needs to be done while the program is running.

Other situations where a dynamic language might be useful are during the development of the programs, where traditionally the developers are involved in the cycle of coding then waiting for the code to be compiled before they can see how the changes affect the program. With a interactive programming environment that also allows for dynamic changes of the program, makes the programmers more productive because they can concentrate on the task at hand instead of having pauses when waiting for the compilation to take place.

3 Solution

What we propose as a solutions is an extension to SELF that will give the programmer a better overview of the changes that are made within the program. The choice for having SELF as a base to extend with our solution is because that SELF is a language that supports some of the properties that we wish to have within our language system, properties are:

- Dynamic (Run-time) compilation
- Prototype-based language
- It allows changes to made during run-time to the prototypes.

The solution we propose adds or changes a couple of concepts in SELF:

- The Serviceport - chapter 5.1 Serviceport
  This is a port through which the programmer can access the running program and its internal structure and behaviour.
- The Object Identifier - chapter 5.2 Object Identifier
  It gives the objects their unique id's when the objects are compiled, it is also used when accessing the program through the serviceport (chapter 5.1 Serviceport)
- The Object Handler - chapter 5.3 Object Handler
  Used to define how the message passing will be performed for the objects it also handles the changes that are requested to the object.
- The run-time compiler - chapter 5.4 Dynamic compilation.
  The run-time compiler has to go through a couple of changes to allow the new type of interactivity that is required within DIL.

4 SELF short introduction

This part of the paper will give an short introduction to how the SELF language works (see also [Ungar & Smith 91] and [Ungar 91]).

One of the fundamental things in SELF is that it features message passing as the fundamental operation, providing access to stored state solely via messages. There are no variables, merely slots containing objects that return themselves. SELF merges state and behaviour: syntactically, method invocation and variable access are indistinguishable - the sender of the message does not know whether the message is implemented as a simple data access or as a method. Since SELF objects access state solely by sending messages, message passing is more fundamental to SELF than to languages with variables.

In Self, there is no direct way to access a variable; instead objects send messages to access data residing in named slots. Since a slot can either contain a state or behaviour there is no distinction of how the message is handled when it accesses the object that is stored within the slot. That means that it is possible exchange one slot that stores a state with a function that returns a random value without making any changes to the rest of the program.

SELF is also based upon the use of prototypes instead of basing it upon the notion of classes. This makes a basic difference in the way that new objects are created. In a class-based language, an object would be created by building it according to a plan in its class. In a prototype-based language the new object is created by cloning (copying) a prototype.

In SELF everything is an object and within the object there are named slots which may store either state or behaviour. If an object receives a message and it has no matching slot, the search continues via the parent pointer. This is how SELF implements inheritance. By implementing inheritance in this way SELF allows objects to share behaviour, which in turn allows the programmer to change the behaviour of many objects with a single change.

The source code is placed in one central location in SELF from where it is retrieved when a function needs to be compiled and the compiled version of the code is placed in a cache from where it is run. If the source code is changed during run-time then the hierarchy that the source code is a part of is invalidated and has to be recompiled.

5 The Dynamic and Interactive Language (DIL)

DIL is based upon the SELF language and has thereby inherited the notion of message passing and behaviour of the objects with theirs slots. The way that DIL is changed from SELF is that it incorporates an object identifier that gives all the objects within the system a unique id, it also adds a object handler to each object that handles the message passing to and from the object (see Figure 1). DIL also changes the behaviour of the

compilation within SELF a little, furthermore it also adds the notion of a serviceport through which the programmer can access the internal structure of the program during run-time.

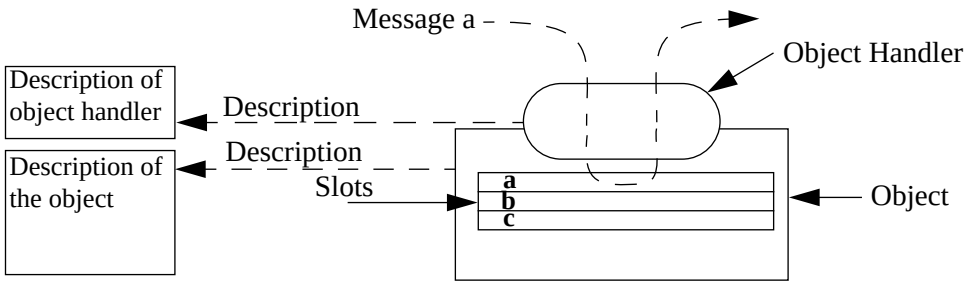


Figure 1. An object and its description

One of the more fundamental changes between SELF and DIL is that in the latter the source code that the program consists of will be encapsulated within the running program so that the only way to access it will be through the serviceport. This is done because the program should be able to keep the source code (description of the objects) up-to-date with the changes that the program itself makes. The descriptions of the objects can be accessed through the serviceport at any time.

5.1 Serviceport

The serviceport introduced within the running programs give an interface through which the programmer can view the internal structure of the program. Initially all the programmer needs to make a running program is to start an empty program and then access it through the serviceport and add the rest of the functionality to the program from there.

How the service-port is connected with the rest of the parts within the program can be seen in Figure 2. It shows the way a change request takes when sent through the serviceport. It begins with that it goes to the object identifier to find out where in the system the object it will affect is located. After that it is sent to that objects object handler that will perform the change and mark the slots that will be recompiled and it will also register the object at the compiler.

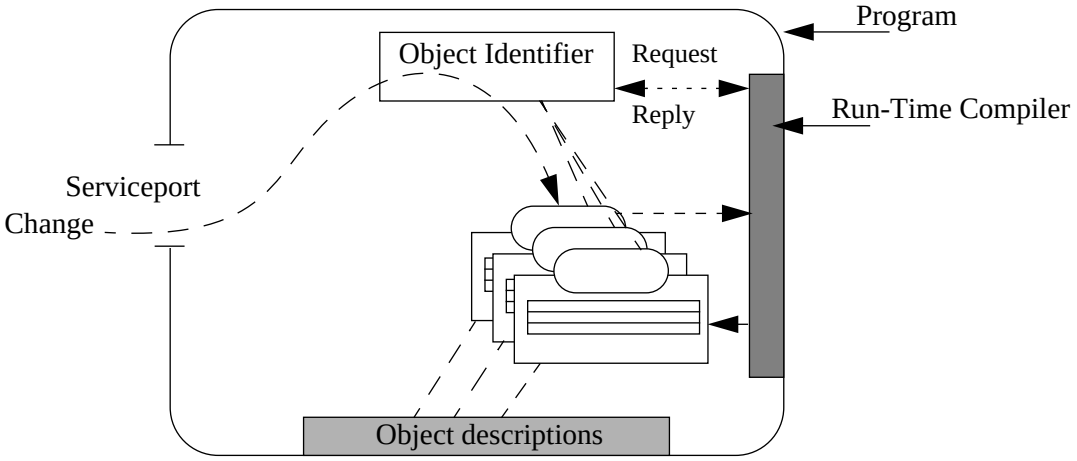


Figure 2. The logical parts of a DIL-program

By having the program encapsulate itself with the description of the source code within the program it is easier for the program to keep track of the changes that are made to the objects.

5.2 Object Identifier

The object identifiers functionality is actually hidden from the programmer all that he will see is an interface that is provided if he access the program through the serviceport.

The object identifier is used for giving every object in the system a unique id, this id is among other things used for connecting the object with its description (i.e. source code). When the object is compiled it is registered with the object identifier and given its id.

The object identifier actually has another task at hand, besides giving all objects an unique id, it works as an interface for the serviceport. This interface gives the user of the service port an overview of the objects that exists within the program. The object identifier passes messages from the service port to the correct object. At the end of the object the object handler takes care of the message.

The methods that the identifier supports in its protocol are rather simple, it can locate an object by its name or it can locate the object by going through a series of links (parental links) to it. It also has the functions necessary for adding new objects and deleting existing objects.

5.3 Object Handler

For handling the dynamic changes that can happen to an object, we have introduced the object manager that handles these request and also keeps the description of the object up-to-date. The description of the object is actually changed when an object has gone through a successful compilation, until then there exists two descriptions of the object, the one that describes the object that is running and one that is showing the programmer how he wishes the object should look after compilation. When the programmer then signals that the changes are done the changed slots within the objects are marked for compilation (see chapter 5.4 Dynamic compilation).

The notion of the object handler is almost the same as for ordinary objects in the way that it also contains slots that describes how the message passing is done. The object handler has a standard way of handling the messages and this is the same way as SELF handles the messages (e.g. the messages are forwarded to the slot with the matching name, if such a name doesn't exists within this object it is sent to its parent where it will be handled or later discovered that there doesn't exist a slot by that name and an error will be generated).

The object handler also contains the behaviour for adding, removing and changing slots in the object. Furthermore the object handler keeps track of the description of the object. The description can be seen as the source code for the object, and it is possible to dump out the description of the object at any time during runtime. This description will then show exactly how the object looked in that moment when the dump was requested. This is useful when the programmer performs some

By having the behaviour of message passing defined within the object handler it is possible to modify the message passing algorithm and implement other ways for passing messages. As an example it is possible to introduce the notion of layers [Bosch 94].

The concept of layers is that the layers encapsulate the object and a message sent to or from the object needs to pass the layers. A layer can intercepts a message and react in a predefined manner to the receipt of the message. This allows for the representation of *relations* by layers. A message that are sent to the object has to pass the layers that encapsulates the object, starting at the outermost layer and working his way in. Each layer receives the message and inspects it in order to determine what to do with the message. The layer can virtually do anything with the message, delay it until some event to occur, change the contents of it, pass it on to another object.

The layers are implemented by a change of the message passing algorithm so that the messages are sent through the layers that are stored within slots in the object. The object handler keeps track of these layers that might be changed in the same way as ordinary behaviours are changed. The only difference from a layer and ordinary slot within an object is that a layer slot can only be accessed by the object handler. In which order the layers will be travelled are kept in track within the object handler.

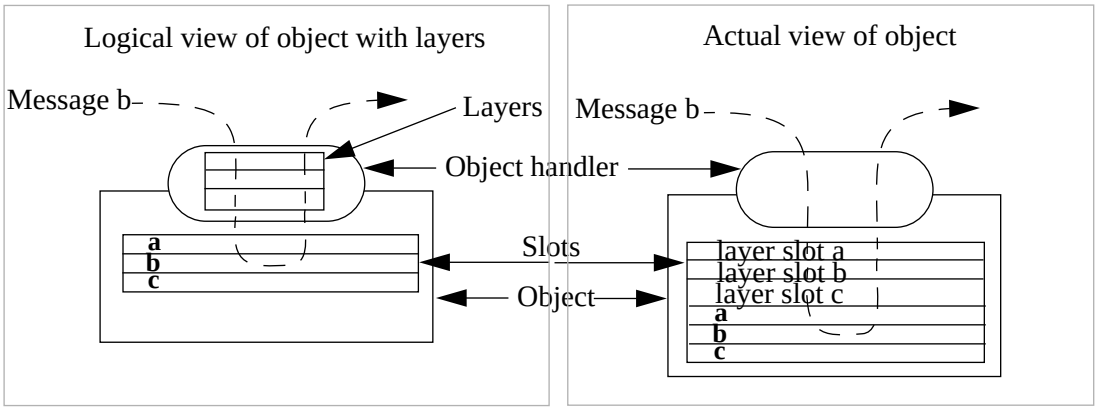


Figure 3. Logical view of the object with layers and the actual view

These type of layers are actually inserted into the object handler within a kind of slots where the first slot is the outermost layer. By having the layers placed within the slots it is possible for the object or the programmer to change the contents of these slots during run-time.

By implementing the concept of layers into DIL makes it will get an even more expressive power.

5.4 Dynamic compilation

DIL deals with compilation in the same way as SELF [Bosch 94] does i.e. dynamic compilation. When a source method is invoked for the first time, it is compiled quickly by a very simple but completely non-optimizing compiler. Conversely whenever the user changes a source method, all compiled code depending on the old definition is invalidated. To accomplish this the system keeps dependency links between source and compiled methods. If the method is executed often it is recompiled with an optimizing compiler.

In DIL it works almost in the same way but since you only can access the description of the objects through the serviceport after you have begun running the program the first time it will work a little different. First of all is that all methods will be compiled as soon as possible after the start-up of the program. The first method that will be compiled is the head function that will start the program and then these that are directly accessed by this method. After that all objects that are created within the running program that aren't compiled will be marked for compilation and they will be compiled as soon as it is possible to do so.

When a change occurs in the program either issued from the programmer through the serviceport or an through the object itself the affected slots where the methods or 'variables' are stored are marked for recompilation. The slots that are marked for recompilation will be recompiled as soon as they're accessed or when the system has time to do the compilation, unless the object that is changed has a thread of control that is running within a slot that has got a request for change, in that case the recompilation is withheld until the execution within the slot is finished, and then the compilation can take place.

Other slots aren't affected by the changes that are made within other slots. Of course will assignment slots to 'variables' disappear when the variable is deleted. This mean that an object that goes through the process of compilation will have the same values as it had before being compiled, except for slots that has been added that will contain a provided value or nil.

6 Inherent problems with dynamic changes

Problems that might occur when having a language that supports changes to be made on both the structural and the behaviour of the objects are that objects that are referred to later are totally removed, another object relied on a certain behaviour that the changed object had before the changes where made and more of these problem when suddenly one object in the dark corner of the program discovers that something has changed and that this weren't good for him.

6.1 Changes in an objects parent

If changes are made to an object hierarchy this might lead to problems for the objects that uses this object as parent. Normally this isn't considered as a problem, since a change to a parent object should also change the behaviour for the parents, but there is one situation that we will show that might lead to problems. Say that we have three objects A, B and C, B and C are children to A (see Figure 4). Object A has three slots that contains behaviour or state. Changing the behaviour of say slot c giving it a new behaviour, will give both objects B and C this new behaviour, unless this behaviour has previously been overridden in the ascendant object as in C, where the change will have no effect.

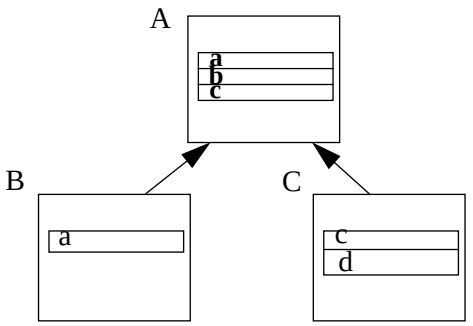


Figure 4. Three objects (object handler removed by purpose)

The problem that might occur is that if we change the behaviour in A method c to the behaviour that is in object B and not removes the override in object C we might have some trouble later when the programmer again change the behaviour of slot c in A and assumes that this will affect the whole hierarchy, because this method is still overridden in object C.

Another thing worth mentioning is that if we decide that object A no longer has the use of c and we remove this the method will still be present in C since it is declared there as an override for the method in A, consequently the function c will no longer be available to object B.

If another object then should try to access the method c in object b when it no longer exists within that part of the hierarchy an error message will be generated and sent back to the object the request originated from (see chapter 6.3 Error Handling).

6.2 Frequently changing a method

Frequently changing a method will lead to problems with recurring compilations and thereby performance problems.

If this changes of the method are merely a swapping between a couple of different version there exists an better approach of doing this. By creating an parent for each of these methods it makes it possible to use the built-in notion of changing the inheritance hierarchy (e.g. just swap to another parent object and when the method is accessed it will be redirected to the correct parent object). Changing the parent object like this

doesn't need any kind of recompilation since the parent object pointer is also stored within a slot and changing parent is therefore just a matter of assigning the new parent to the slot.

6.3 Error Handling

Because of the dynamic nature of the objects the error handling also need to be dynamic so that it can handle the changes that are made to object that might result in errors such as Message Not Understood. For this it might be necessary to add the notion of exception handling within the language. Self actually has some sort of error handling that can be used for this. This sort of error handling works in the way that you can send an additional message with every message that you send, if the first message is not understood or some other error occurred the second method is executed.

7 Conclusion

This kind of dynamic programs will probably become more and more used in the future when the compilation techniques has come that far that the performance will be the closer to those within hybrid languages like C++ that are statically compiled static.

With the help of DIL the programmer will have the power to do powerful programming, this can be both good and bad since DIL allows the most things to happen, it will be up to the programmer to make sure that the program is correctly written. 'If the programmer doesn't know what he is doing then he shouldn't do it.'

The original idea for changing the SELF system into DIL came with the thought about a huge graphical system (cyberspace) in which there exists actors, objects and other types of information that will be running for an eternity after it is started. In this system things can be affected by each other and therefore the need of dynamic behaviours are needed. When thinking in this perspective of time, much of the traditional ways of programming has to be abandoned because they are no longer practical to use. From this big idea the thinking has been scaled down to apply this ideas to the techniques that exists today.

Acknowledgement

A thanks goes to dutchy one (Erik van Campen) for the late night discussions that made me realise that the first approach for this paper wasn't worthless anyway.

References

[Ungar 91]. David Ungar, Craig Chambers, Bay-Wei Chang, Urs Hölzle, 'Organizing Programs Without Classes', LISP AND SYMBOLIC COMPUTATION: An international Journal, 4, 3, 1991.

[Ungar & Smith 91]. David Ungar, Randall B. Smith, 'SELF: The power of Simplicity', LISP AND SYMBOLIC COMPUTATION: An international Journal, 4, 3, 1991.

[Bosch 94]. J. Bosch, 'Relations as Object Model Components,' accepted for publication *Journal of Programming Languages*. Also Research Report 10/94, University of Karlskrona/Ronneby.

[Hölzle & Ungar 94]. U. Hölzle, D. Ungar, 'A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance,' OOPSLA '94 Conference Proceedings, Portland, OR, October 1994.

[Vemulapati 95]. Murali Vemulapati, Ram D. Sriram, Amar Gupta, 'Incremental loading in the persistent C++ language E,' *Journal of Object-Oriented Programming July-August 1995.*

CHIT CR

Magnus C. Ohlsson
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: caesar@pt.hk-r.se
www: http://www.pt.hk-r.se/~mpt95moh/

Abstract
This document contain a proposal for an extension to the C++ language to make it easier to produce real-time systems with concurrency. We discuss the problems concerned when trying to develop real-time systems like the time constraints and memory access. When working with object orientation we also have the problems with inheritance. The reason for choosing to extend C++ is that it is a well spread language and is easy and flexible to work with.

1 Introduction

We have been using object-oriented programming languages for a while now, implementing many different systems. They have been able to protect data and methods through encapsulation and been able to reuse parts of existing system through the possibility of inheritance and the use of classes. Some people like [8] B. Selic and [9] J. Kuusela and M. Awad are working on how to analyse and the design the real-time and concurrency applications but as long as there doesn't exists any support in the object-oriented languages this work has no high priority due to our opinion.

The thoughts about object-oriented systems isn't new. Many others like [6] D. Kafura, [7] J. U. Kiess and [10] C. E. Pereira have looked at the problems involved.

Our approach have been to use the concepts and techniques of object-orientation to define real-time object-oriented systems and in this way reduce complexity through modularization.

1.1 Real-time systems

When we are talking about real-time systems we mean systems that's not only concerned about the in- and output and the correctness of the output but also the time in which the result is send back. There are two types of real-time systems. The systems where the results must be returned within a specified time or the returned value is useless are called hard real-time systems. Those who don't have these constraints, where the computed value isn't useless but a little bit inadequate, are called soft real-time systems. In hard real-time systems the process isn't allowed to continue its execution outside the specified time and in soft real-time systems the process is allowed to continue if it isn't finished.

1.1.1 Tasks

The tasks in a real-time system are either periodical or aperiodical. If a task is periodical it means that the task has to be executed every *n* time unit. To be observed is that the tasks deadline must not be equal to the start of the next execution period. The aperiodical tasks aren't executed periodically. Instead they respond on events.

1.1.2 Scheduling

To keep track of all the processes there is a scheduler included. It handles all the timing constraints, resource requests and priority specifications. Due to all this information it schedules, as optimal as possible, which process that should be executed.

1.2 Concurrency

When we are dealing with real-time systems we also have too deal with concurrency since we have more than one process and these processes are sharing resources and memory. To solve these problems we can use semaphores, monitors, conditional critical regions, event counters or path expressions.

We name this message-based synchronization or sometimes shared-memory.

1.3 Object Orientation

In the object-oriented world we are concerned about some concepts that makes the object-orientation object-oriented.

1.3.1 Class

Classes are identical to types in traditional languages. Later languages permitted the user to define and create new types from other types. No target memory is allocated by creating a new class. This occurs when we instantiated. We can also look at the class as if it was defined by the methods it permits. The class groups and abstracts objects according to their behaviour.

1.3.2 Object

An object is a data structure plus a set of methods. The status of an object evolves through a set of phases, changing its variables or the methods possible to call. Objects can also be seen as autonomous entities with an interface that responds to a set of messages. The message is a request for an object to execute one of its procedures (methods). When an object hides the implementation of its operations and internal data structures from the user it's called encapsulation.

The encapsulation is useful because it helps you hide parts that are not relevant for the user and since the external interface of an object is independent of the objects realization, the reimplementation of an object doesn't affect the other objects in the system.

There exists two kinds of objects when we are dealing with real-time systems. These are:

- Active objects
- Passive objects

The active objects are invoked by messages and also responding with messages. These are implemented as concurrent processes exchanging messages. The passive objects are shared datastructures with some methods associated to change the contents. These are described in [5] Object-Oriented Programs in Real-Time.

1.3.3 Instance

An instance is the object implemented as an area of computer memory allocated to contain the systems data and of what the class represents. It's organized into a data structure as specified by the class definition.

1.3.4 Inheritance

Inheritance is one of the problems when implementing the real-time systems, but more about that later. Inheritance is a structural organization of classes where subtypes take on the default of a superclass or where the class lets it operations and/or local variables is inherited by its subclasses.

Inheritance is performed while defining new classes but no target memory is allocated until an instance of the new type is created.

1.3.5 Overriding

Overriding or overloading of methods is where a single method name invokes one of a collection of identically named methods depending on the types of both arguments and result.

1.3.6 Message passing

When objects communicate they send messages to each other to invoke methods or some other functionality.

2 Advantages

Now we have two concepts, but why do we want to combine them and what can we gain from it?

2.1 Architecture

If we use object-orientation we will have a more natural approach when using software object to model the associated real-world concurrent objects. By doing this we will also achieve a flatter architecture of collaborating objects instead of a top-down hierarchy of functions.

2.2 Complexity

When we today are building real-time systems the systems and the building of the systems are becoming more and more complex. The systems are also growing bigger and bigger. The complexity concerns both the structures and the time domain. Therefore a good approach seems to be the combining of the two concepts to increase the complexity.

The object-oriented computational model is architectural, it can be described as a network of interacting components. This means that the structure reminds of the real-time models structure.

2.3 Modularity

We have also started to realise that to achieve high quality and correctness in their systems it’s is important to be able to specify real-time systems precisely. So with object-orientation we have better control over the modularity and through this the identifiable objects provides a logical partitioning of the problem domain. Unrelated objects can be totally ignorant of each other. Each of the object classes represents a self contained piece of the problem domain which can be examined separately from or within the system.

2.4 Concurrency

The concurrency part are also concerned because the objects are natural units for concurrent execution and allow the real-world concurrency of applications to be naturally modelled.

3 Problems

When we are trying to combine the two concepts, object-oriented programming and real-time systems there will occur some problems we have to deal with. The specification of the problems is provided for a deeper understanding of the problems concerned with real-time and concurrency systems.

3.1 Dynamic systems

Real-time systems are dynamic and dealing with the dynamic time and space demands of real-time systems is difficult. The need for more flexible and intelligent future real-time systems will highlight these difficulties even more.

3.2 Inheritance

The synchronization constraints of an object of and object are defined as a part of its specification and implementation. According to this researchers have highlighted that introducing a new method and/or overriding an inherited method in a subclass requires redefinition. We refer to this problem as the inheritance anomaly.

In [1] Real-Time Specification Inheritance Anomalies and Real-Time Filters the authors have identified and addressed three types of inheritance anomalies. It is mixing real-time specifications with application code, non-polymorphic real-time specifications and orthogonally restricting specifications.

3.2.1 Mixing Real-Time Specifications with Application Code

In cases where we are trying to control how methods are accessed by concurrent users in the form of locking or using some other synchronization device, we have to deal with the code of the method being controlled. The inheritance mechanism is incapable of separating the synchronization code from the non-synchronization code in a method. This means that one can't be altered without redefining the other. That's why it's difficult to leave a method unchanged but change its synchronization or vice versa.

Therefore it may be necessary to loosen the bond between the code which synchronizes a methods execution and the code of a method which are performing the calculations.

3.2.2 Non-polymorphic Real-Time Specifications

This occurs when a real-time specification can't be polymorphically associated with a different method or a set of methods. Then programmers have to define the real-time specifications for every method that requires the same specification. This anomaly also occurs we are going to change the implementation of a method in a subclass and we don't want to change the-real-time specification.

3.2.3 Orthogonally Restricting Real-Time Specifications

If we have many different real-time specifications that are defined independently and combined through inheritance (multiple inheritance may also occur) they affect each other semantically and we can then be forced to redefine some of the related specifications. A common thing is that this anomaly is occurs together with the non-polymorphic specifications anomaly.

3.3 Synchronization

Dealing with real-time system requires synchronization. We have to decide if the object interaction is synchronous or asynchronous. We also have to think about how time is specified in a asynchronous system. A

sub-part of the synchronization problem has already been discussed in ”3.2.1 Mixing Real-Time Specifications with Application Code”.

If we are using synchronous message passing the sender object, that is trying to send a message to another object, will be block until the receiving object is ready to communicate.

If we instead are using asynchronous communication the objects who sends the message will not be blocked even though the receiver aren’t ready to receive the message. With other words the sender can do what ever it want to after the message has been send.

According to [4] L. Bergmans, M. Askit and K. Wakita many languages extend and mix these two communication types to create different concurrency and synchronization semantics. Some of them are:

- Proxies
- Coordinated termination
- Remote procedure calls
- Early returns

The choice between synchronous and asynchronous interactions is not basic. The problem is to choose the model that’s the most suitable for real-time applications. [6] D. Kafura means that it depends on the application and that there are two different types of applications.

In the first one the central problem is the modelling an reasoning about the state of a server and how that server reacts to external events. In the seconds on the central problem is the reasoning and modelling the sequence of events taken by different servers in fulfilling a single function. The first type of systems is better represented in a asynchronous model while the other is better represented by the synchronous model.

As we mentioned earlier it’s possible to mix the two concepts and in the case of object-orientation this is needed because the services provided by a collection of objects is more naturally represented by a sequence while the provision of those services is more naturally represented by a collection of active servers.

The time in a synchronous system is linearly along the sequence of actions but in asynchronous systems there may be more interleaving of events and the ability to reason about time becomes harder.

3.4 Performance

One of the fears we have today is that the systems won’t execute as fast as the need to. The won’t be able to meet their time constraints. Some people mean that experience teaches that decoupled and flexible systems are equal to slow and unpredictable.

We may also have problems with networks because most real-time and concurrent systems are distributed and are therefore dependent on the networks performance. The performance of the networks are also depending on how error-free the communication is. When constructing our systems we have to be aware of this and set our time constraints to a proper level.

Our opinion is that this isn’t a problem any more because the computers have become so much faster and the compiler more efficient. And even if they fast enough now so when we have finished our system in half a year they will be. If we instead are dealing with a slow network We think that the network should be changed to a faster one because sooner or later ordinary real-time systems will need the capacity anyway.

3.5 Maintainability

Object-oriented systems will be hard and difficult to maintain and the components hard to reuse because everything will depend on everything. We can address these problems in functional programming to but the fact that is that deep and complex hierarchies of any kind resits change and therefore are hard to maintain or reuse.

But if the object-oriented approach leads to flatter hierarchies and better modularization the maintainability will be increased.

3.6 Implementation

Later when it comes to the implementation part we will also find some problems. According to [5] J.M Gwinn there are some important aspects on this.

3.6.1 Asynchronous access

Real-time systems require asynchronous access to both active and passive objects because if a system only is build with active objects we are forces to pass input messages around from object to object and updating the messages on the fly because there is no central repository for the relevant information. This will cost in both per-object overhead, CPU overhead and the added context switches. If passive or hybrid objects are present relevant data can be centralized in few such objects.

3.6.2 Memory

When we are developing real-time systems it's a good idea to only use memory blocks with a fixed size. This is done to avoid memory fragmentation. Some systems are providing the use of ones own memory handler.

3.6.3 Message sending

When sending messages from one object to another differs depending on if it's a active or passive object. The syntax form is the one of a subroutine call regardless of it's a passive or an active object. The active objects sends a message asynchronous and the passive object sends the message synchronous. This leads to that the semantics of active and passive objects are different.

4 What other offers

4.1 Composition-filters model

This model is developed by [1] M. Askit, J. Bosch, W. van der Sterren and L. Bergmans to be used with real-time systems and by [4] L. Bergmans, M. Askit and K. Wakita for concurrent systems. The computation model is adopted by the Sina language.

The composition filters model is divided into two parts, which are:

- Interface part
- Implementation part

4.1.1 Interface

The interface part is handling the incoming and outgoing messages, which in turn are handled by input-filters and output-filters.

```
class OrderedCollection interface
     methods
      add(Element) returns nil,
      at(Integer) returns Element;
      remove(Integer) returns Nil;
      size returns Integer;
     conditions
      Initialized;
     inputfilters
      <disp: Dispatch = {Initialized => *};>
end;
```

Example 1. Interface part

The class we see above declares the methods add, at, remove and size to be visible and for manipulating an ordered collection. To invoke these methods we must send an appropriate message to an instance and when it's received it must pass through all the input filters before any execution will take place. An input filter specifies conditions for message acceptance or rejection and determines the appropriate subsequent action.

The filter in the example is of class Dispatch and is used to initiate execution of a method if the message successfully passes it. The condition for the filter is specified between the brackets, "{" and "}". On the left side of the "=>" the condition is specified and in the example it's Initialized. The conditions are similar to logical propositions and are their definition is defined in the implementation part.

The received message is matched with the method names specified on the right side of the "=>". In the example the "*" indicates a don't care condition which means that if the message matches one of the method names in the class OrderedCollection it will be executed.

4.1.2 Implementation

The other part, the implementation part, contains operation definitions, local variables, definition of conditions and an optional initialization operation.

```
class OrderedCollection implementation
     insvars
      collection: Array(100);
      noOfElements: Integer;
      initializedElements: Integer;
     conditions
      Initialized
        begin return initializedElements = 100 end;
     Initial
      begin ... end; //the initial method, which is executed after object creation
     methods
      add(AnElement: Element) begin ... end;
      at(index: Integer) begin ... end;
      remove(index: Integer) begin ... end;
      size begin ... end;
end;
```

Example 2. Implementation part of the OrderedCollection part

The instance variables are declared in the insvars part and are fully encapsulated. Only the methods defined within the object's class are able to access the instance variables directly. The implementation of the condi-

tions are defined by message expression and the structure of the condition implementations are similar to the structure of methods.

4.1.3 Pseudo variables

When dealing with dispatching it's possible to delegate incoming messages to external objects by declaring the target name in the external clause. It's almost the same as delegating to internal objects with the difference that the objects aren't encapsulated within the delegating object.

We have three different pseudo variables which are:

- inner
- self
- server

For example we may have a method named insert() and by using inner.insert() we the method is invoked directly without passing through any filters. If we are using self.insert() the message will be sent to the object executing the expression. In order to invoke an inherited method self must be used as the target of the message. The last pseudo variable is server. So if we invoke server.insert() the message will be send to the outmost instance and will there be dispatched. By using server realizes dynamic binding in that the method insert() may be overridden by a new method insert() defined by the outmost instance in the encapsulation hierarchy, whereas messages to inner and self are statically bound.

4.1.4 Summary

With different filters it's possible to realize different features like synchronization, inheritance, delegation, atomic transactions and multiple views. There's also a special filter for real-time specifications. With the possibility to synchronize we can solve the different anomalies specified earlier. There still exists some issues that hasn't been solved yet like the combination of synchronization filters with other types of filters. One disadvantage maybe that you have to construct all your filter objects to achieve your purposes which can add some overhead in time.

4.2 RTC++

The real-time programming RTC++ is developed by [2] Y. Ishikawa, H. Tokuda and C. W. Clifford. RTC++ is and extended object-oriented model which they have implemented a programming language which extends C++ on the basis of the real-time object model.

RTC++ differ from C++ because it provides active object. If a active object is defined with timing constraints it's called a real-time object.

```
active class O3
{
    private:
     //private data definitions

    public:
     int    m31(char* data, int size) bound(0t30m);
     int    m32(char* data, int size) bound(0t20) timeout(m32_abort);
     int    m33(float f) bound(0t30m);

    activity:
     slave[2]m31(char*, int), m32(char*, int);
     slave   m33(float f);
};
```

Example 3. RTC++ Object

This declaration doesn't differ so much from the C++ declaration except from the keywords active and activity.

4.2.1 Activity

As default an active object has and uses a default thread. It's also possible for a user to specify multiple threads. These threads are called member threads. It's the member threads that are defined in the activity part of the class declaration. It's possible to define two types of threads, slave threads and master threads. A slave thread is an execution unit related to a method or a group of methods. In the example, in the activity part, we can see both a slave thread group executing two methods and a single thread executing on method.

RTC++ are using a priority inheritance protocol in object invocation. This means that the slave thread inherits the priority from the sender. If there exists a queue of messages they are ordered after priority.

```
active class P!
{
     private:
      //private data definition
      void main();

     activity:
      mastermain() cycle(0; 0; 0t200; 0t200);
};
```

Example 4. Periodic task with a master thread

Here we specify a periodic task within an active object. In this example the start-time and end-time are unspecified. The figure after the two zeros are describing with which duration it should be executed and the last figure specifies its deadline.

4.2.2 Timing specification

We can have two timing specifications and where bound is exemplified in the first example. Bound is defining the worst-case execution time and within asserts the deadline time.

4.2.3 Communication

Communication in RTC++ is done synchronous. The syntax is the same syntax as for C++.

```
O3 *v;
// ..
n = v->m31(buf, size);
// ..
```

Example 5. Communication in RTC++

RTC++ is providing two means of sending a reply message, return and reply statements. When we have a return statement a reply message is sent to the sender and the execution of the method is finished. In the other case of a reply statement there is a reply message sent and the subsequent statements are executed instead of finishing the execution of a method.

4.2.4 Summary

RTC++ is useful for both programming hard and soft real-time systems. As described in [1] Real-Time Specification Inheritance Anomalies and Real-Time Filters it's possible to declare the constraints in both

the methods header or in the method body. This means that timing constraints may be visible in the method interface and be encapsulated in the object's implementation. But there is no solution on the inheritance anomalies and there is no real support for concurrency.

4.3 DROL

DROL is created by [3] K. Takashio and M. Tokoro and is like RTC++ also an extension to C++ and it's a distributed real-time system programming language. There are features like time:

- Polymorphic invocation - to realise best effort and worst case execution time.
- Examine mechanisms - to detect timing errors and to realise the property of least suffering.
- Different semantics - for inter-object communication and exception handling as the communication protocol.

4.3.1 Objects

When we are using DROL we define distributed real-time objects. These are defined by declaring three objects, Base Object, AbstractStateMeta Object and ProtocolMeta Object.

4.3.2 Base Objects

The Base Objects are defined by adding the keyword DRObject before the C++ keyword class. It is possible to specify the worst execution time and exceptions. The methods in the objects that are going to be executed periodically are defined by adding the keyword active before the method definition.

```
DRObjects class Aeroplane
{
     private:
      OID       controller;
      APData    current_data;
      int       controlSelf(APData data);
      int       abort();

     public:
      active int master()
        start() end() period(0t05s)
        deadline() timeout(abort);
      landPlane()
        within(0t200ms) timeout();
};
```

Example 6. Base object in DROL

The method master is invoked every fifth second and if it misses its deadline, also five seconds, the method abort is called. The method landPlane has to finish its execution within 200 milliseconds.

4.3.2.1 AbstractStateMeta Objects

These objects is defined by adding the keyword ASMeta before the class declaration. This object is almost like the normal C++ object except that it has one base part and one state part. The Base Objects variables and methods are declared in the base part and in the state part the behaviour concerning the synchronization of arrived messages are defined.

```
partitial_transition_function() {
     operation {
      case action_name:
```

```
        commitment {
           case commit:
              ...
              trans state_name;
           case rejected:
              ...
           case aborted:
        }
        ...
     }
}

ASMeta BBuffer : class BBufferASM {
     ...
     base:
      int  count;
      int  read(char* data, int size);
     state:
      actiondef READ     read(char*, int);
      actiondef WRITE    write(char*, int);
      init statedef EMPTYactionEmpty()
      statedef NORMAL    actionNormal();
      statedef FULL      actionFull():
}

BBufferSM::actionNormal(NORMAL) {
     operation {
      case WRITE:
        commitment {
           case commit:
              if(count == MAX) {
                 trans FULL;
              } else if(count < MAX) {
                 trans NORMAL;
              }
           case rejected:
              trans NORMAL;
           case aborted:
              trans NORMAL;
      }
     }
}
```

Example 7. AbstractStateMeta Objects

By using the keyword actiondef and statedef it's possible to define relations to methods that should be executed in different states. The keyword operations defines which methods that are allowed to be executed in a special state.

4.3.2.2 ProtocolMeta Objects

These objects are defined by adding the keyword PMeta before the class declaration.The definition of ProtocolMeta has a protocol part and the protocols defined are used for inter object communication. Every protocol are specified by using the statements:

- statedef - for state definition
- sendmsg - for send of primitive messages
- receivemsg - for receiving primitive messages
- trans - for state transitions

The first statement, statedef, defines a state and actions to be executed when the protocol transfer the state. The interface to the Base Object and the verification of timing constraints are defined by statements as invokebase and wait.

4.3.3 Summary

DROL has the possibility to describe the semantics of message communication at a meta-level as a communication protocol. This is a good approach to deal with the communication but it can also extract code on which you don't have an good oversight. Some advantages are that timing constraints for messages may only be declared in the methods body and worst case may only be declared in the objects interface and as [1] M. Askit, J. Bosch, W. van der Sterren and L. Bergmans pin-point DROL suffers from the anomaly, mixing real-time specifications with application code.

5 CHIT CR

Some of the problems described above isn't going to be concerned because they are described and sometimes solved in other papers. Most of them do affect our work and had all affected our thoughts.

Our decision to use C++ instead of some other language is based on the convenience that it's one of the most spread languages and it has a lot of flexibility. It also supports many of the OO paradigms or try to do so.

CHIT CR is an abbreviation for C++ High Level Interface to Concurrency and Real-time. The approach we are taking is to extend the language model for C++ and give solution to how this may be implemented so a preprocessor can be constructed which translate the CHIT CR code into clean C++ code.

5.1 Basics

In [2] An Object-Oriented Real-Time Programming Language a system said to be schedulable if it meets all deadlines of a task set. The analysis of schedulability lets a designer of programs under certain conditions predict if the real-time tasks can meet their timing constraints. This requires a bound on the execution time for the tasks. This is possible in CHIT CR.

When defining an active class in CHIT CR it's almost the same as doing it in C++, but there exists some differences. Most of the differences we find in the interface part which means that we are able to avoid the mixing real-time specifications with application code anomaly.

A class can consists of the same things as a normal class like public, private and protected declarations. With CHIT CR it's is also possible to declare a state part.

```
act class C1
{
     private:
      int  v1;
      int  v2;
      char*v3;
      char v4[21];

     public:
      void m1();
      void m2(int, int);
      int  m3(real, real);
      char*m4();

     state:
      normal(nil);
```

```
      writeInt(v1,v2);
      writeChar(v3, v4);
      noWrite(*);
};
```

Example 8. A CHIT CR class

If it's a active class the declaration must begin with act. The class could either be active or a normal passive class. How the synchronization between methods is solved is discussed in section ”5.1.2 Synchronization”.

In the implementation we are doing as in C++ accept that we have to deal with the states.

```
void C1::m2(int a, int b)
{
     (writeInt);
     v1 = a;
     v2 = b;
     (normal)
     cout << “We have now received number ” << v1 << “and “ << v2 << “\n”;
}
```

Example 9. An implementation of a class method

5.1.1 States

We normally look upon states as different sets of methods available at a specific time. In real-time and concurrency applications we have to deal with solutions to avoid inconsistent data. This can happen when we are having many methods concurrently executing and working on the same instance variables.

In example 8 we have defined three states. In the *normal* state it's possible to access all variables. Normal don't have to be declared because it's default. It may be overridden. In state number two and three it isn't possible to access v1 and v2 respectively v3 and v4. This means that when another method or thread has entered some of these states the other methods trying to access them will be blocked and queued until the first method goes into a valid state. The entering of a state is exemplified in is exemplified in example 9.

In the last state, noWrite, the “*” character means that none of the variables are accessible.

5.1.2 Synchronization

In an active class every call to a method results in creating a new thread. It's always the calling side that's creating the thread. With void returning methods (procedures in function oriented programming) this is no problem because we don't have any dependencies with methods waiting for a return value.

In the case of a method waiting for a return value this gets a bit more complicated because if the calling method continue with its executing the value it has received may be very wrong if the called method hasn't finished its executing when the value should be used.

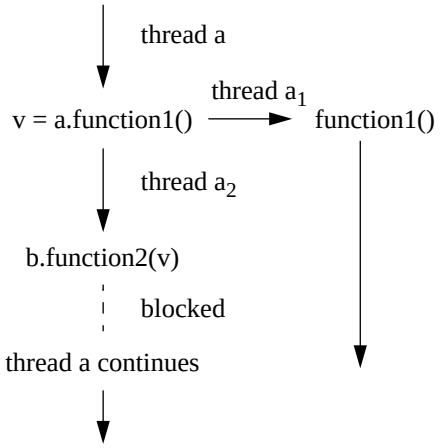


Figure 10. Future blocking

This problem is solved be letting the calling thread continue its execution until the return value is needed. This is called future blocking. In figure 1 thread a is executing and is calling a's function1(). We create a new thread for function1 and the threads starts executing again until thread a_2 has to stop because the return value from function1 hasn't been delivered yet. When it's received the execution continues and thread a_1 are deleted. Instead thread a_2 has to create a new thread for function2 and so it continues.

If a method from a normal class are called from an active method, the call will be synchronously as usual and the calling method will be blocked until the method returns.

5.1.3 Time constraints

To be able to create good real-time systems we have to deal with time constraints. If we can't do this we can't create hard real-time systems. In CHIT CR it's possible to define periodic methods and methods with time constraints.

```
act class C2
{
     private:
      // The private stuff

     public:
      void m5():limit(0.150, clear);
      void m6(int, int):limit(0.50, nil);
      int  m7(real, real):period(0.100, 0.50, 0,0, nil);
      char*m8();
      void clear();

     state:
      // All the states
};
```

Example 11. Periodical and constrained methods

If we want to have constraints on the execution time we use the word limit. The first argument to limit is how long time it has before it must be finished. The time is specified as <seconds.milliseconds>. It's not possible to specify bigger time bounds because our opinion is that if we are dealing with real-time systems the time constraints aren't so big as minutes or hours.

The second argument is a pointer to a function or a method that should be executed if it doesn't meet it's deadline. If we pass nil as an argument it just abort the thread without any special cleaning up.

It's is also possible to specify periodic task. This is done by using the keyword period and it takes five arguments. The first argument specifies how often the method should execute and the second the time within it should have executed. The third and the fourth argument specifies when the method should start its execution and end. The time is specified as <hour.minute.second.millisecond>. The last argument should be a pointer to a function or a method that should be executed if it doesn't meet it's deadline and, just like limit, if we pass nil as an argument it just abort the thread without any special cleaning up.

5.1.4 Blocks

To work our way round the problems with the inheritance anomalies we use something called blocks (we have already started when we only allow real-time specifications in the interface part). A block is a collection of methods with some real-time specification that's the same for all the methods in the block.

```
act class C3
{
     private:
      // The private stuff

     public:
      func:limit(0.200,nil)
      {
        void  m9();
        void  m10(int, int);
      }
      int  m11(real, real);
      char*m12();

     state:
      // All the states
};
```

Example 12. Block syntax

The block could be named, but it's not necessary though it may be a good idea when we are going to inherit from the class. The block name if followed by the keyword limit or period and the parameters for them. All the methods that should be included in the block should be defined between the two "{}".

Now we are free to change the implementation of the methods without changing the real-time specifications and vice versa.

```
act class C4 : public C3
{
     private:
      // The private stuff

     public:
      func:limit(0.100, clear);
      ...
      void clear();

     state:
      // All the states;
};
```

Example 13. Inheritance and overriding

We are also free to change the real-time specification when we have inherited them and we can also override them separately. To be able to do that we have to specify which block the function belongs to. This is because we would have a better overview of the implementation.

```
act class C5 : public C3
{
     private:
      // The private stuff

     public:
      void func.m9(): limit(0.25, nil);
      ...

     state:
      // All the states;
};
```

Example 14. Overriding method from a block

5.2 Implementation

Now we are asking us self how this could be implemented. We are not going so deep into it but instead discuss some ways to solve it.

5.2.1 States

States are like semaphores. When you enter an area you raise the flag to signal that no other should enter the area. The implementation of this could be an invisible object that keep track of the states. When a method is trying to use a variable it must first check which state it's in and then if it's not allowed to access the variable it will be blocked.

Therefore we also need a queue or a queue number to know which methods turn it is. In the state object we could have a queue, linked list, that contains pointers to the methods that should be executed. We might need one queue for every state.

Another solution, and perhaps the easiest, is to use the monitors that the SUN UNIX system provides.

5.2.2 Synchronization

The realization of this are as described before implemented with threads. The future blocking part may be implemented so that there are a while statement, inserted by the preprocessor, checking so that the variable contains a proper value.

5.2.3 Time constraints

Every thread that start execute, that have a time limit, must have some form of clock that keeps track of the executed time. For the C++ language there exists a number of methods that helps you with this but the code for this must probably be included in the implementation of the method.

When we are using periodic methods it's the same thing if we are dealing with time limits except that we have to solve what should be done with the method when it should execute. One solution is to include code that keeps track of when it's time to execute and between there the thread must be blocked. If we have a method with a specific start time we must start the thread immediately and let it be blocked until it's time for execution.

5.2.4 Blocks

The implementation of the blocks are more complicated to describe. Here it's the work of the preprocessor that counts. It has to keep track of all the specifications and extract where needed. How the preprocessor handles this is up to the implementor of it.

6 Conclusions

Implementing real-time and concurrency applications object-oriented is a better way than the old fashion functional way. This is not only true for real-time and concurrency applications but for almost any kind of application. But as we have described there are problems involved. One of the biggest is the problem with inheritance, the inheritance anomaly. The three most common anomalies are:

- Mixing real-time specifications with application code
- Non-polymorphic specifications
- Orthogonally restricting

Our way around this is to work with blocks that have the same type of restriction. With blocks we have loosen the bindings between the real-time specifications and the application code. Another problem is the one with valid data in concurrent systems. The states are our solution on this problem. The states helps us have a better control over our systems.

There still exists some problems that has to be solved. One problem we have is when we are using multiple inheritance. In this case we have to blame C++ for the problems because the conflicts that may occur depends on how C++ is done.

Another thing that we are thinking about is the possibility of using a pool of threads for the objects and like [2] Y. Ishikawa, H. Tokuda and C. W. Clifford restricting how many threads each method can occupy and by this have better control the execution.

7 Acknowledgement

We would like to thank Jan Bosch for his support and the persons involved to check the manuscript. We would also like to thank Maria for her support when writing this paper.

8 References

[1] Author: M. Askit, J. Bosch, W. van der Sterren and L. Bergmans
Title: Real-Time Specification Inheritance Anomalies and Real-Time Filters

[2] Author: Y. Ishikawa, H. Tokuda and C. W. Clifford
Title: An Object-Oriented Real-Time Programming Language

[3] Author: K. Takashio and M. Tokoro
Title: DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems

[4] Author: L. Bergmans, M. Askit and K. Wakita
Title: An Object-Oriented Model for Extensible Concurrent Systems: The Composition-Filters Approach

[5] Author: J.M Gwinn
Title: Object-Oriented Programs in Real-Time

[6] Author: D. Kafura
Title: Issues in the Application of Object-Oriented Techniques to Real-Time Systems

[7] Author: J. U. Kiess
Title: Why is it so difficult to apply object-oriented information system methods to real-time systems besides time constraints

[8] Author: B. Selic
Title: Object-Oriented Concepts for Modelling Distributed Real-Time Systems

[9] Author: J. Kuusela and M. Awad
Title: OMT/RT

[10] Author: C. E. Pereira
Title: On Describing Timing Requirements within a Real-Time Object-Oriented Requirements Engineering Phase

Pursuing the Minimal Type System[1]

Henrik Hermansson
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: mpt95hhe@pt.hk-r.se
www: http://www.pt.hk-r.se/~mpt95hhe

Abstract
When programming using conventional object-oriented programming languages you are constantly forced to think a lot about the data types of the operands in expressions. The existing languages without this property are 'un-typed' and are checked at run-time. This leads to late discovery of errors (if at all) and performance reduction. We want to create a type system that requires little or no explicit type information and gives the programmer great flexibility, while still incorporating the usual properties of statically checked type systems. This can be achieved by an extension of the type checking scheme used in Standard ML. A method in this new type system do not restrict the argument types with it's interface. If a data type is accepted by the method code, it is also accepted by the method interface. This type system allow the programmer to use a method as generically as possible.

Keywords: type system, object-orientation, programming language, programming

1 Introduction

This paper is based on the idea that current object-oriented programming languages are not sufficiently easy to use. We will only deal with how the type system and it's effect on code degrades the efficiency of the programmer. We describe the basic need for a type system and how this should preferably be represented in an object-oriented programming language. The paper will also try to find the core type system properties that should be kept. There is also a proposition for a new type system which will give much flexibility with the minimal amount of explicit type information. This type system is only outlined here in it's most basic form; a complete proposal is not within the scope of this paper.

2 Programming distractions

The task of the software developer is to describe the problem-domain in terms of the constructs available in the used programming language. The programmers description is then translated into a machine executable language. What the developer should concentrate on is to describe/model the application domain as accurately as possible. Unfortunately existing programming languages and execution environments are too restricted to fully enable this. The developer has to spend time doing optimizations, saving memory and fighting the type system instead of making good models. This does not only take time but also, which is worse, moves the developers focus from the model to the technical aspects of compilers and interpreters.

To solve this problem the need for the low-level work has to be reduced, i.e. we need to remove the source for the focusing on memory usage, type compatibility and such. Preferably it should be removed not only from the language constructs but to some extent also from the mind of the developer. This paper will only treat the problem of the type system and it's need for programmer involvement. It will also describe why we still need have a type system.

---

1. This paper is part of the proceedings of the first Advanced Object Oriented Concepts (AOOC) mini conference at the University of Karlskrona/Ronneby, January 1996.

Much of today's research is focused on how to bring static and dynamic type systems together, taking the benefits of both, i.e. make static typing more flexible and dynamic typing easier to error-check and bind statically. The solutions proposed by these researchers is often to add a certain language construct to deal with the problem. This is generally not a good solution, since it adds technical complexity to the language and often lack uniformity.

What we rather would like to see is uniform concepts that, at least partly, changes the objectives of the programmer. This is however not an easy thing to do; when raising the level of abstraction in a programming languages the result often becomes too restrictive or too complex for productive usage. Thus we believe that the current programming levels, in general, has to be kept. What we instead do is streamlining the existing language concepts (in this case the type system) and support it, often by the use of tools.

The type system is something that easily becomes a 'fact of life'. Once the limitations are known they are accepted as unavoidable and nothing as done to improve the 'facts'. The type system is something that also affects the style of programming, although it is not required. Most programmers claim to program object-oriented, but when examining the resulting code it differs a lot merely depending on the language used (e.g. C++ code VS SmallTalk code). This difference is, in part, a result of differing type systems. Thus the type system affects the programming style.

3 Reasons for a type system

Even though the types in a programming language are disturbing there is a number of reasons for having them. The fundamental reasons will be described here and in "Where to reduce type information" on page 6 we will see how the type system can be simplified with respect to these aspects.

3.1 Describing data

The ability to describe data is fundamental in every programming language and the initial reason for data types. We simply need some way of telling what kind of data an expression is using. The data types is therefore the primitive base of every language. Types is just as important in object-oriented languages, since it is an efficient and natural way to describe sets of data. Object-orientation has made this less 'basic' in a sense, since usage of abstract data types can modify the data descriptions of language.

3.2 Error detection

Types and type checking is a good error detection tool. Type checking prevents programs from performing operations and methods on objects of incompatible data sets, such as multiplying two strings.

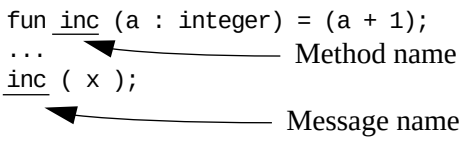
```
integer i; bool b;
i = i * b;
```

In the expression `i * b` the operands `i` and `b` are checked to be of types which are compatible with the multiplication operator `*`. The assignment thereafter is also checked in a similar manner.

The checking involved with types can occur at any time before the expression is evaluated. In practice, the time of checking make an important division of today's type systems, which we will see later.

3.3 Determining message receiver

Type checking is used to determine the matching of messages and methods.



In the above example[1] we see a method declaration and a message send (a method call).

In the type system the question is ‘which method in which object to call’ for each message send. The matching can be made either at run-time or at compile time. SmallTalk uses the run-time binding, also known as dynamic binding. Conceptually it works like:

1. A message is sent to an object (method call).
2. It is examined for type information.
3. If the name and type of the message matches that of any of the objects methods, they are bound (i.e. the method is invoked).
4. If not, the message is sent on to the objects ancestors.

If static binding is used then all the message-to-method connections are made at compile time.

The matching is done based on the name and type of the massage, and the name and type of the methods. If only the names were used, invalid connections could be made (e.g. with wrong number of arguments) and method/operator overloading would not be possible.

When the language incorporate overloading several methods can have the same name, but differ on the number and types of arguments. The matching must use the type of the arguments.

3.4 Documentation

Apart from being used by the compiler/interpreter, the type information can be used as a sort of documentation. To explicitly indicate the type of e.g. a return value of a method is helpful to the user of the that method. If no type information is in the method declaration the programmer is dependent on good comments or other sorts of documentation.

4 Concepts of current type systems

This chapter will describe some of the properties of a few existing type systems. The type system is characterized by the language it is used in.

4.1 Type inference

Type inference is the process of determining the type of an expression by examining it’s context. This is a notable feature of ML, which is described more in “Standard ML” on page 5. With type inference the type of an argument can be deduced by analysing the way it is used. This is not possible in every case, sometime

1. Throughout this paper we will use a ‘Standard ML’-like syntax to describe type systems. See [Wikström 87] for information on the ML syntax.

a little help is needed by the programmer. Experience from ML indicates that the type inference can handle most cases remarkably well [Wikström 87].

4.2 Static type checking

Static type checking is the validation of type correctness *before* execution time. In general, this means that the checking of data types is performed at compile and/or link time.

To enable static checking every expression must have a type, i.e. it must be explicitly stated by the programmer or deduced by means of type inference, as discussed above.

4.3 Dynamic type checking

Dynamic type checking means that no type correctness is ensured until run-time. This actually mean that an expression is without type; it is it's resulting *value* that has a certain type. When the value of an expression is incompatible with the involved operation the type checker will halt the execution.

Dynamic type checking has the advantage of allowing more flexible language constructs. An often used example is the ability to store and retrieve objects of differing types in a generic way.

There are two major disadvantages with dynamic type checking. The first is that it is performed for each expression executed, so every possible program flow actually has to be tested for a program to be guaranteed type correctness. The second problem is that since it is performed at run-time there is an overhead involved both for performance and memory.

4.4 Type equivalence

When type checking is performed the types are often checked for equivalence. There are two major definitions for type equivalence;

Structural equivalence. Mean that the types T_1 and T_2 is equal if and only if they can carry the same set of values. This is especially useful when comparing constructed types like in the following example.

```
type x = {integer,real}
type y = {integer,real}
```

In this case the types `x` and `y` would be considered equal even though they have different names.

Name equivalence. $T_1 \equiv T_2$ if and only if T_1 and T_2 where defined with the same name. With name equivalence the example types `x` and `y` would *not* be considered equal.

4.5 Polymorphism

Polymorphism is a common feature of object-oriented languages. Polymorphism mean that the type of an object can vary within a certain range, often it can only be a subtype of the declared type. Since the exact type of the object is not know until execution the method to bind to a message sent to a polymorphic object can not be known at compile-time. Thus run-time (dynamic) binding is needed, but compile-time (static) type checking can still be used.

4.6 Method overloading

Method overloading is when the message-method connection is determined by *both* the name and arguments, as mentioned earlier.

5 Some existing type systems

When designing a programming language in general and the type system in particular, a number trade-offs has to be made, often between performance and flexibility. These trade-offs give the language it's nature and character.

We will describe the type systems and their features of some characteristic programming languages.

5.1 C++

Strong static checking, dynamic binding

C++ is a language which incorporates both static and dynamic binding, complete static type checking and name type equivalence.

The static checking and simple dynamic binding gives C++ high execution speed and no run-time errors. The dynamic binding is also checked statically.

C++ requires considerable amount of explicit type information from the programmer. C++ has also 'inherited' explicit coercions (called 'cast') from C. These can be used to delude the type checking system and to make programs break for strange reasons.

The latest draft (april 95) of the C++ standard have added a few constructs for simple, but effective, run-time type checking.

5.2 SmallTalk

Typeless, dynamic checking, dynamic binding

In SmallTalk a variable has no type, it is the value stored in it that has. This has two advantages: high flexibility and no explicit type information. Unfortunately it also has a drawback: since any variable can hold any type of object, the type checking and binding has to be performed at run-time.

SmallTalk is indeed without explicit type information and no one will complain if you call a strange method. But this is paid dearly when the errors occur - at run-time. We consider this a too high price to pay for flexibility and 'code freedom'. This actually gives us too much flexibility; we only want to be able to make 'sensible' calls flexible.

5.3 Standard ML

Static checking using type inference, polymorphism without dynamic binding

Standard ML (Meta-Language) is a functional programming language with an interesting type system. The ML type system infers almost all types. ML only need explicit type information where the type of a method can not be exclusively determined by it's implementation.

```
fun even (n) =
   (n mod 2 = 0)
```

In the above example the type of the method is inferred using the following steps.

1. The operator `mod` has the type '`integer` $\times$ `integer` $\longrightarrow$ `integer`', thus `n` must be of type `integer`.
2. Since '`n mod 2`' resulted in an integer the equality test for integers is the next type expression; '`integer` $\times$ `integer` $\longrightarrow$ `boolean`'.

3. This is the last expression in the `even` method so the type of the method is '`integer` $\rightarrow$ `boolean`'.

If there had been another `mod` operator with type '`real` $\times$ `integer` $\rightarrow$ `integer`', the type inference would have failed and required explicit type information about `n`.

If the argument `n` had not been used at all, it would have been a polytype (i.e. any possible type) and the method would had worked on any type of object (polymorphism). This is the notion of polymorphism in ML.

5.4 Cecil

Flexible static checking, static and dynamic binding

Much of Cecil's flexibility comes from the multiple dispatching (multi methods), which allows message matching not only on the receiver of the message but also on *all* the arguments run-time types. Even though message-method matching depends on the run-time types the type systems manages to check the possible types statically.

6 Where to reduce type information

It is fairly obvious what kind of type system we would like to have; all the properties mentioned in "Reasons for a type system" on page 2, but with as little as possible disturbance due to type information. It is a great advantage to perform the type checking and binding early. Lets take the features one at a time and see if the benefits can be kept while the unnecessary type 'thinking' is reduced.

6.1 Describing data

As a means for describing data, types can not be removed. It is definitely needed when abstract data types are composed out of 'smaller' types. Note however that explicit information is not needed to express the usage of the data later.

6.2 Error detection

As seen in ML the amount of explicit type information can be reduced to a minimum and still statically checked. If we accept the performance overhead of dynamic checking (like SmallTalk) then we can check the types without any explicit type information.

6.3 Determining message receiver

To connect a message send with a method, the names of the message and methods is enough. If we also want method overloading, we need type information about the arguments for the method and the message.

Polymorphism complicates the type system even further; it is impossible to statically bind messages to polymorphic objects, thus some type information has to be carried at run-time to determine the actual type of the object.

The need for type information does not imply that the information is explicit, it can be inferred or determined at run-time.

6.4 Documentation

The documentation reason for having explicit types can be satisfied by commenting the code. When having comments instead of inflexible explicit type information the comments can be rich where needed (e.g. the interface of public methods) and sparse where not needed (e.g. encapsulated methods).

7 The minimal type system

Since we are not entirely happy with the type systems shown in “Some existing type systems” on page 5 we define a new type system. This system is based on the one used in Standard ML, but we have made the type for each expression weaker; each ML-type is turned into a set of possible types.

The reason for this type system is that we want to keep the flexibility close to that of SmallTalk and have the early checking and binding (as often as possible) of e.g. C++.

The first two objectives (flexibility and early checking) can always be achieved by our solution. The early binding, however, can not always be used.

7.1 How it works

Here is a simple example of how the type checking works:

```
fun sum (x,y) =
   (x + y);
```

If the operator + is overloaded to be either of type ‘`integer × integer ⟶ integer`’ or ‘`real × real ⟶ real`’ then, in ML, we have a type clash. What we do with our type system is to let the type of the method `sum` be the set of possible types, based on the type inference done within the method. Thus the type of `sum` becomes

```
{ integer × integer ⟶ integer | real × real ⟶ real }
```

meaning that a call with either two integers or two reals will be allowed. As with most object-oriented systems we allow the actual types to be sub-classes of the required types.

Let’s look at another example (not very ‘ML’-ish in syntax, though):

```
// Swap, ver. 1

fun swap (x,y) = (
   var tmp = x;
   x = y;
   y = tmp;
);
```

Let us sat that from previous declarations we know that the following ‘=’ operators exist:

```
{ integer × integer ⟶ nil | real × real ⟶ nil | char × char ⟶ nil }
```

To determine the type of the `swap` method the type inference is applied roughly like:

For each data type in the program:

1. ‘`x`’ assigned that type.
2. If the type matches the expression ‘`tmp = x`’, then proceed.

3. If the type matches the expression ‘`x = y`’, then let `y`’s type be the one used for `y` in the matching method (for the assignment operator it is the same as for ‘`x`’) and then proceed.

What this means is that every type is tested on argument ‘`x`’. All data types that pass through the whole method are possible. The constraints on the other method arguments are discovered during this process.

The resulting type for `swap` becomes ‘`{integer × integer ⟶ nil | real × real ⟶ nil | char × char ⟶ nil}`’.

If the method code would have looked like:

```
// Swap, ver. 2

   fun swap (x,y) = (
      var tmp = x;
      x = sqrt(y);
      y = tmp;
   );
```

and the method ‘`sqrt`’ had the type ‘`{real ⟶ real}`’. Now the type for ‘`x`’ would be constrained to only include type ‘`real`’. This constraint, in turn, make the method type become ‘`{real × real ⟶ nil}`’.

With this system we can have our methods as generic as possible. This is a result of that the only discriminating factor on the method argument’s type is how the argument is used later. Note that it is not only how each individual argument is used, but also how they are related to each other, through the method code. If this were not true, calls for the `swap` method (version 1) like the following would have been allowed:

```
real r; integer i;
swap( r, i );
```

The determination of the method type, start at the ‘lowest’ method level, i.e. methods that only use built-methods or operators. Then all used methods are type determined using the process outlined below.

1. Each method argument initially has the set of all types.
2. For each data type in the program: the first argument is assigned this type and the method is stepped through to check if the type can be used in every expression (where the first argument is used). If it can, the other types required on the other method arguments are noted.
3. When all possible types have been tested on the first argument, the type of the method is: the set of all sets of argument types that matched every expression. Each return type associated with the argument types is connected to the method type.

This solution has flexible constructs allowing ‘as generic as possible’ methods. The only big drawback is the dynamic binding (run-time dispatching) needed, due to the lack of precise type information. This can not be avoided, but it can be optimized to minimize the performance impact. The techniques used in the SELF compiler [Ungar & Smith 91] has proved that dynamic binding can be optimized with good results. In those cases where the set of type matches for a method call is the singleton set, static binding can of course be used.

Table 1. The connection between the type set and the method binding

Cardinality of the matching type set	**Resulting binding**
0	Type error, no match
1	Single match, static binding
>1	Multiple match, dynamic binding

One could argue that the use of type checking is diminished. This is somewhat true, but it is diminished in such a way that it will not be a great disadvantage to an experienced programmer. The type checking is still there, if an argument is used in two expressions of different types, there is a type clash. The complete program is actually checked to ensure that no SmallTalk-like ‘messageNotUnderstood’ errors will occur in run-time.

Note also that in the `swap` method the variable declaration for `tmp` is without explicit type. The type for the variable is determined in a manner similar to the argument determination. If the a data type is specified anyway, it will constrain the expressions and types around it.

There is a possibility of very big type sets for some types of programs, resulting in long compilation times. We hope that this will not be problem in practise, it has to be tested in normal use to be verified.

To have separate compilation with this type systems, can be a problem. This has not been looked into yet.

7.2 How do this reduce the type distractions?

This type system requires almost never any explicit type information (in fact it requires less type information than ML). The only situation where it might be needed is to solve ambiguities. This allow the programmer to concentrate on the vital parts, doing the right thing (as opposed to doing the thing right, which the type system hopefully will make sure).

So the explicit type information is reduced and the flexibility is kept. It is not quite as flexible as SmallTalk but much more flexible than previous statically checked languages.

Since methods will be as generic as possible, a programmer will not have to write special versions of the method to satisfy the type system. If it should ever be desired, however, it is easily done by inserting an explicit type discriminator.

References

[Apple 92] Andrew W. Apple, “A Critique of Standard ML”. Princeton University, 1992.

[Chambers & Leavens 95] Craig Chambers and Gary T. Leavens, “Typechecking and Modules for Multi-Methods”. 1995.

[Duggan 95] Dominic Duggan, “Dynamic Types Have Existential Type”. Department of Computer Science, University of Waterloo, Canada, 1995.

[Lee 91] Peter Lee, “Topics in Advanced Language Implementation”. ISBN 0-262-12151-4, Massachusetts Institute of Technology, 1995.

[Ungar & Smith 91] David Ungar and Randall B. Smith, “SELF: The Power of Simplicity”. LISP and Symbolic Computation, 1991.

[Watt 90] David A. Watt, “Programming Language Concepts and Paradigms”. Prentice Hall, 1990.

[Wikström 87] Åke Wikström, “Functional Programming Using Standard ML”. Prentice Hall, 1987.

Garbage collection in C++ using smart-pointers

Jonas Matton
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: mpt95jma@pt.hk-r.se
www: http://www.pt.hk-r.se/~mpt95jma

Abstract
One of the largest sources of errors in software products today is memory management. One way to reduce the problems concerning allocation and deallocation of memory is to use a language that supports the concept of garbage collection. Although the notion of garbage collection is old, quite a few of the languages today are from the time when it wasn't feasible to the performance of the language to use it. In this paper we will use smart-pointers to introduce garbage collection into C++.

1 Introduction

A frequently used object-oriented programming language these days are C++ [ANS95] [Str91]. It's a hybrid language since it inherits everything from the C language [Ker78] and then extends it in order to support the object-oriented paradigm. The C language was initially designed for constructing and porting the UNIX[1] operating-system and therefore it allows you to construct programs of a very low-level kind. All of this is also possible in C++. However the world is constantly changing and today we use C++ to program all kinds of applications, everything from operating systems to word-processors. If we are programming an operating-system it's desirable to have total control of the program execution and in this case we might not want to use garbage collection since it introduces unpredictable execution of the garbage collecting program. However if we aren't writing a program with hard constraints (operating-system, real-time systems etc.) then we would like to have garbage collection to help us reduce the errors of allocation and deallocation of memory. Typical examples of applications where we have the most benefit of using garbage collection are those that allocate and deallocate lots of objects like graphical user interfaces and drawing tools. Programs that uses tree structures and others that has complicated pointer structures are also beneficial of garbage collection.

The approach in this paper is to extend C++ with a few classes that provide it with garbage collection behaviour. We will present an approach that lets us use an optional garbage collector i.e. if we need it we use it, if we don't need it we just use C++ as is.

Throughout the paper OMT [Rum91] and C++ examples are provided. It's desirable to have some knowledge in these areas in order to fully understand everything.

In the paper the word 'he' is used when we mean 'he or she'. This is for convenience reasons only.

This paper is constructed as follows; chapter 2 describes which alternatives of applying garbage collection into C++ we have. Chapter 3 describes what a smart-pointer is and how it works in C++. Chapter 4 explains how the reference counting garbage collection algorithm works. Chapter 5 describes how to use smart-pointers to realize the reference counting algorithm in C++. Finally chapter 6 sums it all up with a conclusion.

1. UNIX is a registred trademark of AT&T.

2 Garbage collection in C++

In this chapter we describe several ways of how to incorporate garbage collection into C++. We look at their advantages and disadvantages.

2.1 Incorporating garbage collection as part of the language

How should we incorporate garbage collection into C++? One way would be to implement a compilator that does all the work and provide us with a run-time system that deals with the disposal of garbage objects. There are advantages and drawbacks of this approach as we shall present here.

Advantages:

- Efficiency, at the compiler level we know more about the objects internal structure which can be used to implement an efficient solution.
- Transparency for the user of the C++ compiler. He doesn't even have to know it's there.

Drawbacks:

- It's a side-step from the draft ANSI C++ standard [ANS95].
- A specialized C++ compiler must be used.

2.2 Implementing garbage collection as part of the application space (specialized garbage collecting)

All of us who ever has written a program in C++ knows that dealing with allocating and freeing memory with the use of pointers is a major source of errors. Very often when writing larger C++ programs the programmer implements some specialized way of dealing with such problems. This could be looked upon as a specialized way of applying garbage collection. A typical example of having garbage collection as a part of the application space is to use a class-library that provides it. The proposed solution later in this paper uses this technique.

Advantages:

- We get a more controlled way of reducing errors due to memory allocation and deallocation.
- We have the flexibility to leave the garbage collection out or even change it if it doesn't live up to our expectations.

Drawbacks:

- Introduces overhead in time while collecting the garbage.
- Lacks standardization. Everyone has its own garbage collection.
- Often imposes information in the users objects.
- Risk of misuse, mixing "real" pointer and smart-pointers.

3 Smart pointers in C++

As the title of the paper reveals, part of the solution to our problems has to do with something called smart-pointers. So what is a smart-pointer? It's exactly what the name implies. It's a pointer construct which provides us with the possibility to execute a piece of code every time a pointer is referenced.

In C++ the smart-pointer is implemented as a regular non-static class. The smart-pointer class must overload the method operator->. When the object is called through this method we can intercept object references and thus program the behaviour we need. A typical declaration of a smart-pointer in C++ looks like this:

```
class SmartPointer
{

public:
   SmartPointer(MyType* aPtr);
   ~SmartPointer();
   MyType* operator->();

private:
   MyType* MyTypePointer;

};
```

In the example above which shows the class declaration of a class that implements the operator-> method. This is not yet useful since you must specify exactly what types you want to implement the operator-> method for. If we instead implement the smart-pointer class with generic type (as a template) it becomes more interesting. All of a sudden we can instantiate smart-pointers of any type. This is what the class declaration looks like:

```
template<class T> class SmartPointer
{
public:
   SmartPointer(T* ptr);
   ~SmartPointer();
   T* operator->();

private:
   T *TemplateRealPointer;
};
```

What we do here is to override operator-> method, the implementation looks likes this:

```
T* SmartPointer :: operator->()
{
   return TemplateRealPointer;
}
```

This will provide the smart-pointer class the possibility to act as a regular C++ pointer. In order to make full use of this is to implement behaviour in assignment and other operators to make is suit the certain problem we are trying to address. We shall see how this works with garbage collection later.

4 Reference counting garbage collection algorithm

One of the most straightforward techniques of garbage collection is called reference counting. The reference counting algorithm works as follows. Each user-defined object contains a reference count variable. When a pointer to the object is created the reference count variable is increased. When a pointer is assigned to point to another object the old object's reference count variable is decreased and the new object's reference count variable is increased. When a pointer is destroyed, the reference counter is decreased. When a reference count variable reaches zero it is safe to remove the object and reclaim the memory since no object will be able to access it if it doesn't have a connection to it. A check to see if the reference variable is zero is made every time the reference count variable is decreased.

Figure 1 visualizes how reference counting works. The two pointer objects point to a user defined object. The object contain the reference counting variable.

Figure 1. Reference counting

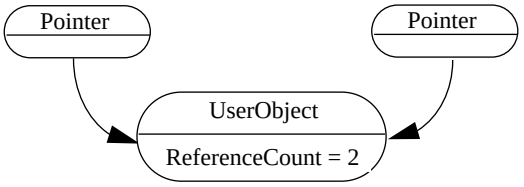
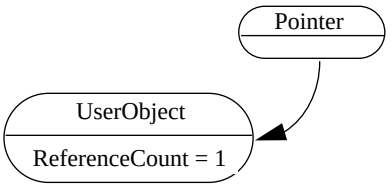


Figure 2 shows what has happened after one of the pointer objects has been destroyed. As we can see the reference count variable is subtracted by one and the pointer object is removed.

Figure 2. Reference counting



There are problems with the reference counting algorithm. First the work that the garbage collection algorithm does is proportional with how much the application program does. For each new pointer that is created or destroyed garbage collection is executed. Second, when a reference count variable reaches zero the object is deallocated. If the object that is deallocated contains further pointers that are destroyed and it can become a recursive deallocation of objects. This is applicable to large structures of “part-of” relations in an object-oriented program.

5 Reference counting garbage collection in C++ using smart-pointers

5.1 General description

The topics described above, reference counting garbage collection and smart-pointers in C++ can be merged with each other in order to provide C++ with garbage collection capabilities. The approach is to make it as transparent as possible for the user of garbage collection system, but of course since it’s not part of the language itself it does impose some things for the programmer which want to use garbage collection. The two main things that the user must be aware of are:

- Change all regular C++ pointers into the supplied smart-pointer class.
- For each object that the user creates he must publicly inherit from a special class that contains the reference counting behaviour for each object.

5.2 The class SmartPointer

We shall now see how to apply the smart-pointer described earlier in this paper in order to provide garbage collection behaviour in our C++ programs. First we shall take a look at the SmartPointer class declaration.

```
template<class T> class SmartPointer
{
public:
   SmartPointer();
   SmartPointer(T* ptr);
```

```
   SmartPointer(const SmartPointer<T>& aSP);
   ~SmartPointer();
   T* operator->();
   SmartPointer<T>& operator=(SmartPointer<T>& aT);
   SmartPointer<T>& operator=(T* aT);
   T& operator*();

private:
   T *TemplateRealPointer;
};
```

This is what the class declaration of the class SmartPointer looks like.

```
SmartPointer :: SmartPointer()
{
   TemplateRealPointer = NULL;
}
```

The constructor with no parameters only sets the “real” pointer to the defined value NULL. This indicates that it doesn’t point to anything yet.

```
SmartPointer :: SmartPointer(T* ptr)
{
   TemplateRealPointer = ptr;
   ptr->increaseReferenceCount();
}
```

To be able to use the class SmartPointer we provide a constructor that takes a “real” pointer as an argument.

```
   SmartPointer :: SmartPointer(const SmartPointer<T>& aSP)
   {
      aSP.TemplateRealPointer->increaseReferenceCount();
      TemplateRealPointer = aSP.TemplateRealPointer;
   }
```

A copy-constructor is provided to handle situations like passing a SmartPointer as an argument to a method or when assigning object at instantiation.

```
SmartPointer :: ~SmartPointer()
{
   if(TemplateRealPointer != NULL)
   {
      TemplateRealPointer->decreaseReferenceCount();
      if(TemplateRealPointer->referenceCount() == 0)
      {
         delete TemplateRealPointer;
         TemplateRealPointer = NULL;
      }
   }
}
```

The destructor of the smart-pointer has an important role in the garbage collection. It first check to see if the smart-pointer has a “real” pointer assigned to the smart-pointer. If it does the object pointed to by the “real” pointer calls the decreaseReferenceCount method. This will decrease the reference count by one. Then it checks if the objects reference count is zero. If it is the object is deallocated since no one uses it any more. After that the “real” pointer assigns the value NULL to indicate no reference.

```
T* SmartPointer :: operator->()
{
   return TemplateRealPointer;
}
```

The operator-> only returns the “real” pointer. This provides the smart-pointer with pointer behaviour similar to a “real” pointer.

```
SmartPointer<T>& operator=(SmartPointer<T>& myT)
{
   if(TemplateRealPointer != NULL)
   {
      TemplateRealPointer->decreaseReferenceCount();
      if(TemplateRealPointer->referenceCount() == 0)
      {
         delete TemplateRealPointer;
         TemplateRealPointer = NULL;
      }
      myT.TemplateRealPointer->increaseReferenceCount();
      TemplateRealPointer = myT.TemplateRealPointer;
   }
   return *this;
}
```

The operator= is the big garbage collection engine. First a check is made to see if the “real” pointer value isn’t NULL. If the pointer has the value NULL that means that no reference is currently stored in the smart-pointer. The SmartPointer is currently unused. If it’s NULL the “real” pointer in the SmartPointer supplied as argument increase its reference count variable and the “real” pointer is copied to our currently Smart-Pointer.

If the “real” pointer wasn’t NULL we are doing a reassign of a previously used SmartPointer. In this case we decrease the old “real” pointer and check for a reference count of zero. If it is we deallocate the object pointed to. After this we increase the reference count of the object pointed to by the new “real” pointer. We also copy the “real” pointer to the current SmartPointer.

```
SmartPointer<T>& operator=(T* aTPtr)
{
   if(TemplateRealPointer != NULL)
   {
      TemplateRealPointer->decreaseReferenceCount();
      if(TemplateRealPointer->referenceCount() == 0)
      {
         delete TemplateRealPointer;
         TemplateRealPointer = NULL;
      }
      aTPtr->increaseReferenceCount();
      TemplateRealPointer = aTPtr;
   }
   return *this;
}
```

This is exactly like the other operator= method with the difference that this one take a “real” pointer as an argument instead of a smart-pointer. This is made because we would like to be able to assign a smart-pointer with a “real” pointer.

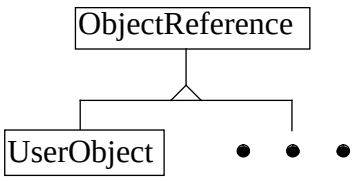
```
T& SmartPointer :: operator*()
{
   return *TemplateRealPointer;
}
```

Once again to provide the smart-pointer with “real” pointer behaviour we implement the operator* method which returns the object itself.

5.3 The class ObjectReference

For each user-created object we must add a little header with the reference counting information and the ways to increase, decrease and retrieve the reference count. The way this is done is to make it a super-class of all the user constructed classes. It is made this way because it imposes the least effort for the user, he only has to publicly inherit from the class and then never bother again.

Figure 3. ObjectReference



We call the class ObjectReference and it's a very simple class. Here is an example of how the declaration of class ObjectReference looks like:

```
class ObjectReference
{
public:
   usigned int referenceCount() const;
   void increaseReferenceCount();
   void decreaseReferenceCount();
   virtual ~ObjectReference();
protected:
   ObjectReference();
private:
   unsigned int referenceCountValue;
};
```

As we can see the constructor is declared as a protected member. This will guarantee that an object of that class can only be created by a publicly inherited sub-class. This make it impossible to instantiate an object of class ObjectReference itself because we will get errors during compilation if we try. Instead we must publicly inherit from the ObjectReference class. This will encourage the user to inherit the class. Also the destructor is declared to be virtual. This will guarantee that the destructors in will be called from the bottom and up in an inheritance tree.

The methods of the class is described here:

```
ObjectReference :: ObjectReference()
{
  referenceCountValue = 0;
}
```

The reference count is set to zero when an object is created.

```
unsigned int ObjectReference :: referenceCount() const
{
  return referenceCountValue;
}
```

The referenceCount method make it possible for the SmartPointer class to read how many references to the object that exists.

```
void ObjectReference :: increaseReferenceCount()
{
  ++referenceCountValue;
}
```

The increase method just increases the reference count by one.

```
void  ObjectReference :: decreaseReferenceCount()
{
  --referenceCountValue;
}
```

The decrease method just decreases the reference count by one.

5.4 A user example

Here is an example of a simple program that uses the classes described earlier. It looks like this:

```
#include <iostream.h>
#include "MyObject.hh"
#include "SmartPointer.hh"

int main()
{
   SmartPointer<MyObject> a(new MyObject);

   a->doSomethingUseful();
   return 0;
}
```

The difference here is that we use a SmartPointer instead of a “real” pointer. Note that we don’t have to do an explicit delete on the allocated object, this is all handled when the destructor of the SmartPointer is executed at the end of the program.

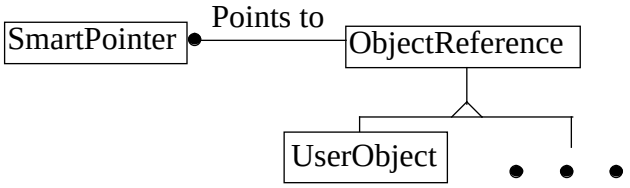
```
class MyObject :: public ObjectReference
{
public:
   MyObject();
   ~MyObject();
   void doSomethingUseful();
};
```

The class MyObject is different from a regular user defined object due to the public inherit from the ObjectReference class.

5.5 System design

Most likely the application that uses this way of garbage collection will look like this:

Figure 4. Typical application



What this picture is trying to explain is that several SmartPointer objects can point to the same ObjectReferenc object. The class ObjectRefernece is the super-class of all user defined objects in the application.

5.6 Existing problems

The presented solution works nicely but it still has some problems associated with it. A great problem is that it's possible to mix "real" pointers with smart-pointers. This introduces the kind of problem that of constructing statements like:

```
SmartPointer<MyClass>* SP_Ptr = new SmartPointer<MyClass>;
```

This is everything we didn't want it to be. A "real" pointer to a smart-pointer and the smart-pointer points to some object.

6 Conclusion

For most applications it's desirable to use garbage collection as a way of reducing the problems with memory allocation and deallocations. However there are situations where we want to have an option if we want to use garbage collection or not. This is especially important in a language like C++ that by inheritance from C has a reputation of being a language to develop low-level software like operating systems in.

We have tried to provide garbage collection functionality as a part of the application space and still keeping the transparency for the user to the greatest extent possible.

As the software products becomes bigger and bigger it's not acceptable to find errors of the allocation/deallocation kind. An application cannot crash just because we had problems handling the memory correctly. This will become even more crucial in the future when we see more and more systems that cannot accept a restart of the software at any time.

Acknowledgement

I would like to thank my colleagues at the mpt95 study program at college/university of Karlskrona/Ronneby for taking the time to discuss this issue with me and provided me with different views of problems and solutions.

I also would like to thank Jan Bosch for support and always providing an answer to my questions.

References

[ANS95] 'Working Paper for Draft Proposed International Standard for Information Systems— Programming Language C++', X3J16/95– 0087 X3, INFORMATION PROCESSING SYSTEMS WG21/ N0687. 28 April 1995.

[Ker78] Brian Kernighan and Dennis Ritchie: 'The C programming language'. Prentice-Hall. Englewood Cliffs. NJ. 1978. ISBN 0-13-110163-3

[Rum91] James Rumbaugh ... [et al], 'Object Modelling Technique'. Prentice-Hall. 1991. ISBN 0-13-629841-9.

[Str91] Bjarne Stroustrup, 'The C++ programming language' 2nd edition. Addison-Wesley. 1991. ISBN 0-201-53992-6.

[Str94] Bjarne Stroustrup, 'The Design and Evolution of C++'. Addison Wesley. 1994. ISBN 0-201-54330-3.

[Wil92] Paul R. Wilson: 'Uniprocessor Garbage Collection Techniques'. University of Texas. 1992.

Definition Modelling

Magnus Robertsson
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: mpt95mro@pt.hk-r.se
www: http://www.hk-r.se/~mpt95mro.html

Abstract
This paper describes the problem with traditional programming languages and their relation to the analysis and design. An approach towards language extensibility is discussed and evaluated. We will see that traditional extensible programming languages suffer from intuitive thinking and can not cope with new concepts in a desirable manner. A solution, the definition modelling language, is presented as an approach towards concept programming, i.e. to define the problem domain and adopt the programming language to the specific problem, instead of vice versa.

1 Introduction

1.1 Background

The attractive thing about object-oriented languages is that they give an easier, more natural way of designing systems. The evolution of programming languages is that they more and more provide features to express concepts of the real world. In today's object-oriented languages we have concepts like

- Objects: a concluded word for things we can identify in the real world. This is used to abstract a system in a logical manner.
- Relations: to represent relations between objects.
- Inheritance: some objects can inherit from another objects. This is used to structure and reusing objects.

However, these concepts do not provide all of the concepts we would like to have. There always arise problems how to express the domain with the object-oriented paradigm. For example the number of relations between two object is not sufficient for expressing certain concepts. Process control is one example of the lack of relations. Another example is an object that affects, i.e. have a relation to, all other objects. There exist no explicit way of handling such a relation.

Recent work and research aims at extending the language in order to extend the object-oriented paradigm. We will see that they are not intuitive, i.e. have no concepts that can be found in the real world, and that they only provide a better way for designing new compilers. It is important to have the expressiveness in the language and not just an interface to the compiler.

The basic thing is that we want to solve a problem in a domain where the new domain often introduces new concepts. These concepts can be hard for the developer to cope with, due to the programming language. A solution would be to express the new domain within the language, i.e. make the new concepts a part of the language.

2 Development methods

When we develop a system we try to translate a real world problem into structured executable and language specific components. The object-oriented development method is divided into three stages: *analysis*, *design* and *implementation*. In the analysis phase we try to specify the users needs in an understandable way [Aksit & Bergmans 92]. In the design phase we extend the analysis to be realized in some way. In the implementation phase we convert the design into a physical system.

The analysis, design and implementation phases can be seen as a translation between the real world and the physical system, see figure 1.

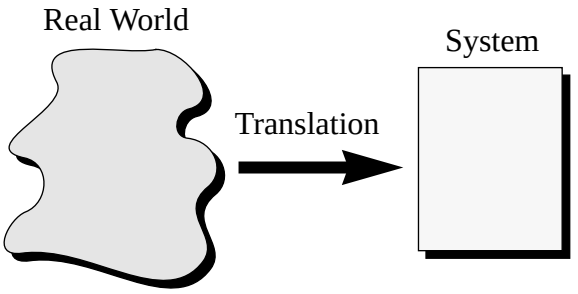


Figure 1. The translation problem exist o a system.

There exist many types of object-oriented development methods (Booch, Top-down, Coad & Yourdon, Jonson & Foote, OMT, Responsibility-driven, Role-analysis). They all consist of a number of activities and phases for creation of models/graphs that represent the real world.

2.1 Analysis

Translation from the real world to an analysis model can be tricky. The expressiveness of the analysis method is crucial since you must be able to express new concepts within the analysis domain. For example, an object-oriented programming language like C++ is more easier when implementing an object-oriented design than a language that don't, for example C. By having features in the language that supports the concepts in the design it is easier to solve the problem.

However, problems occur when the real world consist of new concepts that the analysis method does not support it. Traditionally by adding objects and relations we can solve these problems, but there will always be a limit where we will have an over-complex object structure. [Aksit & Bergmans 92] have been identifying this problem.

2.2 Design

Translation from analysis to design introduces some problems. First is the phase language specific, i.e. a language affects the result of the phase. This leads to inconsistency to the analysis and the direct tractability disappears. We will be talking more about these problems later in the paper.

2.3 Implementation

The implementation of the design is totally language specific. If the language does not support the real world concepts, the code will be totally twisted since the concept will be spread around in the code, see Figure 3.

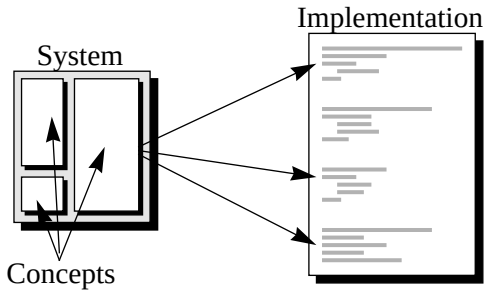


Figure 2. The concepts in the system are spread over different places in the implementation.

2.4 Conclusion

There exist several problem with today's development methods. The translations between the different phases makes a fuss in an unmanageable mess.

To solve this problem many developers tries to design systems into big frameworks where we can easily change and add objects to adopt the concept (the framework) to the specific problem. However building frameworks are not that simple and the result isn't always great. If we could use the concepts in the programming language instead we would have our framework from the beginning.

3 Problem statement

3.1 Introduction

In the previous chapter it was concluded that the problem with developing software products laid in the translation part. We can translate a problem fairly good into an analysis but when translating the analysis to a design and furthermore to a physical implementation we have to be more programming language specific.

This is not good since the programming language is then the limit of how much we can cope with. For example, in theory can we develop anything in machine code, but the size and the complexity it would bring sets a limit of what we could develop.

3.2 Design problems

The problem we have identified are the following ones:

- No one-to-one relation from the real world and the implementation.
- Not enough concepts in traditional programming languages.
- No satisfactory support for dynamic behaviour.

One important thing is that requirements states the problem and the design is a solution to that problem. But the concepts should exist in both the requirements and design. Unfortunately, will we see that the concepts in the design are hard to find and often spread over the system.

3.2.1 No one-to-one relations

If we follow a concept form the real world to the implementation we will see that the concept often is spread around the code. There exist no one-to-one relation between the concepts and the implementation, see Figure 3.

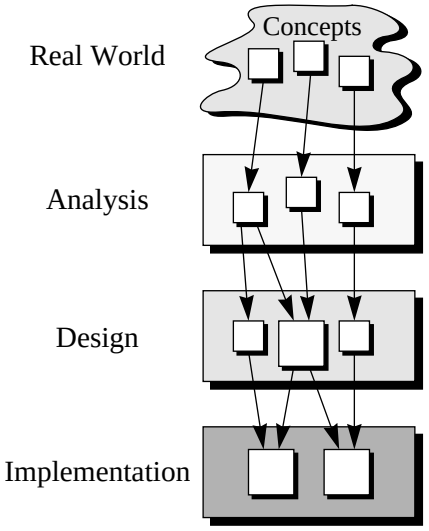


Figure 3. The concepts have no one-to-one relation to the implementation. The concepts are divided or merged into new concepts that are not intuitive or traceable.

This is not good since if we want to change the concept or add more concepts we will have to change in more than one place in the code. This leads to bad traceability since there will be no intuitive feeling of what the code really came from. There is many examples where this problem can be identified. The reason why it is so hard to maintain a product is dependent on the problems with the traceability.

3.2.2 Lack of concepts

Since there in no one-to-one relation we believe there is a lack of concepts in the traditionally programming languages. When we try to express something in a programming language we often have to divide a concept into different parts as explained before. In an object-oriented programming language we only have three different concepts/relations, *inheritance*, *part-of* and *association*. The last is implemented by references. These three concepts is not enough for expressing all concepts in the real world. This leads to the one-to-one relation problem. These two problems walk hand in hand in the object-oriented paradigm.

3.2.3 Dynamic behaviour

Object-oriented development methods of today have no model for expressing dynamic behaviour in an satisfactory way. They only provide some sort of state-chart diagrams which is more or less a state finite machine. What we would need is a method and a language where we can express the dynamic behaviour.

3.3 Today programming languages

A big problem is that the programming languages today, is not intuitive. With intuitive we mean that they must have features to easy express a certain concept within the language. For example to express a process control system in a language like C++ isn’t intuitive. This is because of the lack of expressiveness as described in chapter XX.

The evolution of programming languages have been that they are built on a lower level language with some extra additional concepts, see the first figure in Figure 4. This makes the languages static and hence the lack of expressiveness follows.

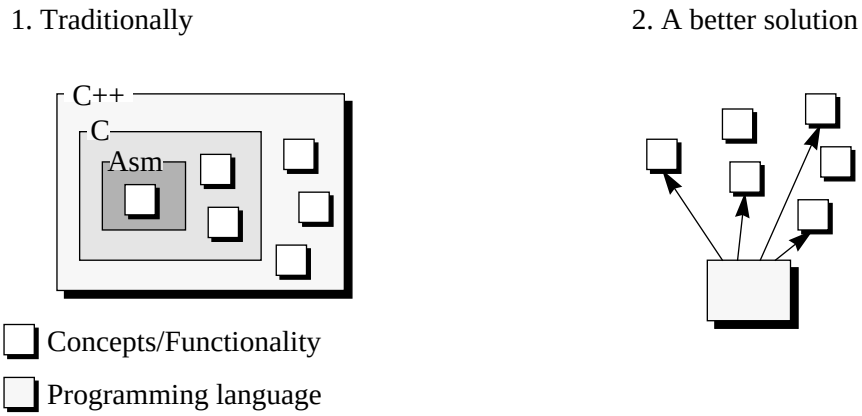


Figure 4. Each programming language includes a static set of concepts that can be expressed (1). A better solution would be to express the concepts within the language (2).

A better solution would be to have a language expressing the concepts, the second figure in Figure 4. The concepts would then be a part of the language. In some cases the developers develop a totally new language due to the fact that the languages today do not provide the features (concepts) that is required for the specific task.

Instead, we could have a library with the most common concepts. These concepts can be reused so when we want solve a problem we can build the language on basic concepts. An example can be the inherit concept. If we run into a concept that can not be described within the language; totally new concepts can expressed in the language.

3.4 Object-oriented anomalies

How can we say that the object-oriented paradigm is the most suitable for a specific problem? For each new system built we adopt the development method to meet the requirements for it. For example, if we are developing a reliable real-time system, we put more effort in design and increasing the test phase to meet the requirements. But what we do not do, is to change the programming language, since it is static. This is not good since we will have to blend all concepts into a big mishmash.

3.5 Extensibility problems

3.5.1 Ambiguity

A problem with extensible programming languages is that they can be ambiguous. The ambiguity comes when we creates new concepts that contradicts the other concepts, or more often, changes a concept.

3.5.2 Concepts cooperativeness

There is also a problem of how the different concepts will work with each other. For example if we specify a concept real-time and then add persistence and concurrent. This leads to a complex system where we must specify what will be in favour of the rest. Is is to meet the real-time constraints or to have the object persistence? This is similar to the ambiguity problem, since the two concepts counteract with each other. When we “blend” two different concept we must specify how they will work with each other. We can not

just assume that they will cooperate just because we have specified them. We believe that this issue is the most difficult part of language extensibility.

4 Related work

4.1 Introduction

There is several researchers who have seen the problem of language incompatibility to the real world [Aksit & Bergmans 92]. Some developers have accepted these facts and tries instead improve their development methods in order to compensate the poor productivity.

4.2 Reflection

There exist two models of reflections. Structural reflection and computational reflection. Structural reflection concerns the reflective use of classes and meta-classes for implementing objects. A meta-class is a class for an instance. Structural reflection allows for extension of the static part of object-oriented languages [Ferber 89].

Computational reflection is the ability to reason about itself [Maes 87]. It is based on the fact that each object has its own meta-object that represent its structure and its way of handling messages. This is very good for *implementing description*, *monitor features*, *interfacing* and *debugging* and *self-reorganization*.

Computational reflection is often implemented by meta-objects. A meta-object is an object that describe and handles an another object. For example the message-handling process for an object can be controlled by its meta-object. However, meta-objects is not the only way to implement computational reflection. It is also possible to reify the communication process, i.e. every message is described as an object.

4.3 L^{AY}OM

4.3.1 Paradigm extensibility

In [Bosch 95] paradigm extensibility is defined as a process of extending the object-oriented paradigm with new concepts. The fundamental idea here is to extend the paradigm used in an object-oriented language to meet the new concepts introduced in the design. Furthermore [Bosch 95] says that paradigm extensibility is the central element of software development and that the language model and method are reflections of this paradigm. We totally agree with this statement but there exist very few programming languages that provides paradigm extensibility.

4.3.2 L^{AY}OM

As an solution to this problem, [Bosch 95] introduces the layered object model (L^{AY}OM). It provides paradigm extensibility through *layers* and compiler extensibility. The layers are then used for concepts like inheritance and part-of relations. As a demonstration for its extensibility new relations, for example partial inheritance is presented. Partial inheritance is used when we only want to inherit specific parts of an object.

The *layers* can be seen as a bunch of transparent paper where each paper represent a new concept. By adding a new paper we have extended the system without touching the previous one. If we look above all

papers we can see the interface for an object without looking at the separate concepts. It is like lifting the concepts in a new dimension.

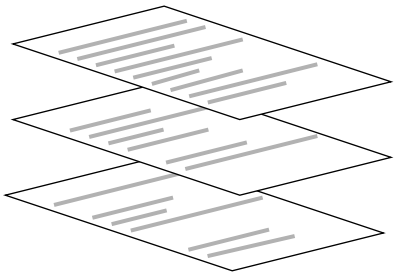


Figure 5. The layered object model can be seen as a bunch of transparent papers.

The layers works like a meta-communication model where each message is parsed within each layer. The layer can then discard, change or redirect the message. It is close related to computational reflection where each layer is an meta-object.

We do not think that this approach is intuitive. The layers does not exist in any form in reality and therefore references in the real world are hard to find.

Furthermore when we add a layer we must extend the compiler to make the layer work. The solution can critically be seen as an advanced yacc-compiler where we have communication between the language and the compiler.

The fundamental idea is good though; extend a programming language with new concepts. However, this approach does not solve the underlaying problem as stated in chapter XX. To minimise the distance from the real world to a executable system.

5 Definition modelling

5.1 Introduction

In our problem statement, chapter 3, we identified the following problems:

- Translation problems from real world to implementation. There exist no one-to-one relation between real world concepts and the implementation.
- Lack of concepts in traditionally programming languages.
- Today’s extensible languages are not intuitive in their arrangement.
- Existence of ambiguity in extensible languages.
- Cooperation problems between concepts.

We have also found, from the previous chapter, that these problems could not be satisfactory solved in today languages.

Our solution to this, is the definition modelling. The definition modelling are a modelling technique where we define all the concepts in a specific domain. The language, the definition language, have some few primitives used for expressing concept within the language itself.

5.2 Implementation

To be able to express concepts within the language itself, we must either, (1) make a new compiler for each extended part of the language. LayOM Bosch 95 is such a solution, and earlier we saw that this was not a desirable solution. Instead we must, (2) have a recursive compiler that changes itself while compiling. It is preferable to make all the concept descriptions in the beginning of the code, since the compiler would be less complex.

A concept in the definition modelling has certain abilities, for example to distribute. It is like each concept has an activity, a responsibility to follow. For example an object is managed by a concept. We will see how this works in a further example.

5.3 Fundamental primitives

To have a language that can describe new concepts we must have some fundamental primitives that can describe the concepts. We have been identified the following primitives:

- Concept - the language must be able to define new concepts.
- Time - the language must have some time features since concepts can depend on time.
- Encapsulation - the language must handle concepts that contain of several concepts.

Objects do not exist as in ordinary object-oriented languages. An object in the definition language is all primitives (key words). So, an object as object-orientation describes it, is not an object until we have defined the concept for it. Then the object is managed by the concept of an object.

Since, there exist no primitives is there no typing in the definition language, unless you define it for the compiler.

5.3.1 Key words

The following key words are used within the language:

- *define* - *as* - to define a concept as something.
- *encapsulate* - an object that contains several objects.
- *change* - represent changes to a state in an object.
- *execute* - represent an object that can be executed.
- *data* - represent an object as data in some form.
- *concept* - the object is a concept.
- *for* - key word for representing an object inside an object.
- *relation* - represent a relation between objects.

5.4 Definition Language

This is only a prototype of how we would like an extensible language to work. Since the language itself have no functionality like ordinary programming languages we will have to define some fundamental concepts, see Figure 6.

```
// Defines Method as concept where the concept can be executed.
define Method as concept ( execute Method ).
```

```
// Defines Variable as concept of being data.
define Variable as concept ( data Variable ).

// Defines Object as concept where the concept encapsulates the concepts Variable and Method.
define Object as concept ( Object encapsulate Variable, Method ).
```

Figure 6. Fundamental definitions.

The inheritance concept comes automatic since we can define a key word as an Object, see Figure 7.

```
// Definition for Person.
define Person as Object.
define Name for Person as Variable.

// We will automatic inherit Clerk from Person since we have define it as the same thing.
define Clerk as Person.
define Book for Clerk as Variable.
```

Figure 7. Inheritance through out is-a definition.

If we change the Person we will implicit change the Clerk.

After these concepts can we define part-of relation. See Figure 8.

```
// Define Part-Of as concept with the writing "Object A Part-Of Object B" where
// B encapsulates C.
define Part-Of as concept "Object A is Part-Of Object B" ( B encapsulate A ).

define Car as Object.
define Tire as Object.

// Define the Part-Of relation for the Tire.
Tire is Part-Of Car.
```

Figure 8. Part-Of relation i the definition language.

If we have an example where a controller controls a process we can do it as Figure 9. The process and controller are both transformers since they both transforms something.

```
// Define Control
define Control as concept "Object A Control Variable V1 with Variable V2" ( change V1 ).

// Define the transformer.
define Transformer as Object.

// Definition of Process and Controller as transformers.
define Process, Controller as Transformer.

// Define a process P.
define P as Process.
// Define the variables Water and Giver for P.
define Water, Giver as Variable for P.

// Definition of C.
define C as Controller.

// Assign the concept Control to P and C.
C Control Water for P with Giver for P.
```

Figure 9. The process and controller example.

5.5 Final words about definition modelling

This language is not supposed to be seen as a clean solution. It is very hard to define a language that can support total expressiveness. We believe though, that an approach towards extensible programming languages is more preferable than standing still in the evolution and think that object-orientation will solve everything.

The definition language is more a taste of the extreme in extensibility programming languages. The definition language can be seen as an advanced yacc-compiler where we have a simpler interface than yacc. Not the most perfect picture of an programming language.

6 Conclusion

6.1 Summary

We have seen that there exist several problems in both the object-oriented development methods and the object-oriented languages.

An approach, the definition modelling, where we define the concepts before using them, where presented and evaluated. It is difficult to define extensible programming languages.

At last, the trouble with lack of concepts is not only a programming language problem. It is also a trouble for the analyse and design phase. It is hard to identify the problem domain structure and requires much domain knowledge.

Acknowledgement

Many of these ideas have been brought up and discussed under curses in mpt95. Without these discussions, the ideas in this paper would never been born.

And finally, a thanks to Lisa, mum and dad and beer.

References

[Bosch 95] J. Bosch, ‘Layered Object Model - Investigating Paradigm Extensibility,’ *Ph.D. dissertation*, Department of Computer Science, Lund University, November 1995.

[Aksit & Bergmans 92] M. Aksit & L. Bergmans, ‘Obstacles in Object-Oriented Software Development’, *OOPSLA 92*, Department of Computer Science, University of Twente, 1992.

[Ferber 89] J. Ferber, ‘Computational Reflection in Class based Object Oriented Languages’, *Research Report LAFORIA*, Université Paris, France, 1989.

[Maes 87] P. Maes, ‘Concepts and experiments in computational reflection’, *OOPSLA-87*, October, 1987.